GPU-based rendering to a multiview display

Verburg, E.I.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

GPU-based Rendering to a Multiview Display

By
E.I. Verburg

Supervisors:

dr. C. Huizing (TU/e)
ir. G.T.G. Volleberg (Philips)

Eindhoven, June 2006

Abstract

This thesis is the result of a project on rendering to a nine-view lenticular display device of two megapixels. We successfully implemented a render engine on a personal computer equipped with a programmable video card.

The render algorithm is based on a technique called micropolygon displacement mapping, which is suitable for generating mutually disparate views. The solution allows for real-time playback of disk-streamed RGBD video sequences. These sources contain both color and depth information at a standard definition resolution.

(This page intentionally contains only one sentence.)

Summary

This thesis presents a method for real-time rendering to a multiview display device given a 2D video source with depth information. Although the method is suitable for various types of multiview displays, this project specifically targets a nine-view lenticular display. Our implementation uses a personal computer equipped with a programmable graphics programming unit (GPU).

The conversion process from the before mentioned source format to a video sequence that is suitable for a lenticular display can be divided in two steps. First a set of mutually disparate views needs to be generated based on depth, and second, these views have to be merged by multiplexing the appropriate subpixels.

Regarding the first step we adopt a rendering technique called micropolygon displacement mapping to create eight additional views. Traditionally this technique enables feature manipulation of 3D rendered objects. However, we exploit it to generate views which simulate a virtual array of cameras. With respect to the point of view provided by the input video, four cameras represent distinct positions on the left side and the other four represent positions on the right side. The views corresponding to each side are computed in a single rendering pass as follows. First, the vertices of a highly tessellated planar triangle grid are manipulated in z-direction based on the depth map. Each quadrant of this grid is manipulated by a different amount, but receives the same color frame as texture map. A specific orthogonal projection of this textured geometry results in four distinct views with horizontal disparity. This pass mainly involves the vertex shader hardware of a GPU.

The next step is to multiplex the subpixels of these preprocessed views into a single lenticular image. This step can be mapped conveniently on pixel shader hardware. For each output pixel we determine the view number of the red, the green and the blue subpixel. Given this number we then fetch the appropriate color values from the textures in video memory.

The engine that implements this algorithm for nine-view rendering has been tested on a system with NVIDIA's 7800 GTX graphics processor. It enables a throughput of 25 frames per second at 1920 × 1080 output resolution when using a triangle grid of 692 × 520 cells. The number of cells determines the quality of the perceived depth impression. Ideally this grid would contain twice the number of pixels of an input frame, but the current GPUs are not yet capable of processing larger amounts of data. This undersampling of the depth map does not harm the depth impression much. Occlusions are supported, but de-occlusion areas are only handled by linearly interpolating the colors of the surrounding pixels. Future GPUs will enable higher frame rates or better quality and are likely to support even a higher number of views.

(This page intentionally contains only one sentence.)

Table of Contents

Acknowledgement

Hereby, I thank the people who helped me during my graduation in one way or another, especially my mentor Guido Volleberg, who gave me this tremendous opportunity to do a project at Philips Applied Technologies. His continuous enthusiasm and sincerity, I will never forget. I also thank Kees Huizing, Bettina Speckmann and Jack van Wijk from the Eindhoven University of Technology for their time and effort.

And not least of all, let me express my appreciation to my family, friends and colleagues for their support; with whom I got to spend a lovely time and had many invaluable discussions, both professionally and socially. It has been a great experience to work in the field of video processing and even write my first paper in addition to this thesis.

(This page intentionally contains only one sentence.)

1 Introduction

1.1 Project and Assignment

Multiview display devices are emerging, but suitable media sources are still scarce as broadcasters continue to deliver two-dimensional (2D) video. Various conversion algorithms are now being developed in order to provide a three-dimensional (3D) experience from different kinds of media.

On the other hand, programmable graphics processing units (GPUs) have evolved into powerful image processing devices over the last few years. These devices for personal computer (PC) platforms can be used to aid the before mentioned conversion.

A lenticular display is a specific type of multiview display device. Its name originates from the small lenses contained on the surface of the screen. These lenses redirect each view in such a way that it enables stereoscopic vision without the use of 3D glasses or other headgear.

Currently, one operational set-up is built from a personal computer (PC), a Field-Programmable Gate Array (FPGA) and a lenticular display. An FPGA is a programmable logic chip. It is programmed to convert an input signal (RGBD) to an output signal (RGB), which is fed into the lenticular display. RGBD denotes a video stream composed of the colors red, green and blue, and a depth channel. This depth channel stores the relative distances between the camera and the objects being captured, as shown in figure 1.

Due to recent developments in computer graphics hardware the graduation project 'GPU-based rendering to a multiview display' was initiated. A GPU is the primary processor of video cards found in PCs. We designed an algorithm for multiview rendering to a lenticular display and implemented it on such a graphics processor. The resulting application is able to process video in real-time as required. This helps the future development of 2D video to RGBD conversion by enabling faster prototyping of the produced sequences.

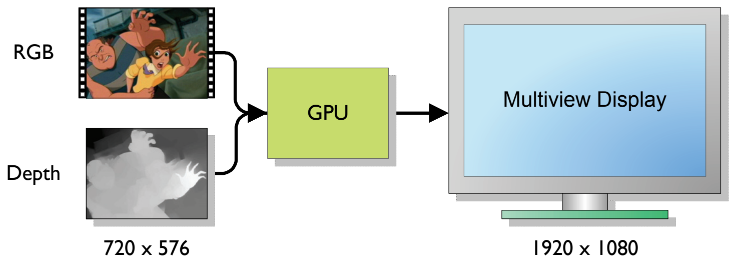

Figure 1. A programmable GPU processes a video stream containing color and depth information to drive a multiview display device.

1.2 Document Structure

The remainder of this document is organized as follows.

In chapter 2 we describe the parts of the common architecture of both PC and video card that are relevant to our use case. For each component the minimal requirements to sustain real-time video playback are defined.

In chapter 3 we provide an overview of traditional 3D rendering techniques related to texture mapping. The focus is on how these candidate techniques can alter surface details of virtual objects. The discussed techniques form a prerequisite for the design concepts discussed in the next chapter.

Chapter 4 first presents the specifics of multiview rendering. Thereafter two adapted algorithms are presented for multiview rendering; both tailored to the architecture of a programmable GPU. We also propose a method, which enables the detection of the inherent artifacts of the first algorithm.

Chapter 5 discusses topics on GPU programming, such as shader models and shader languages, and describes the software implementation details of our render engine.

The project results are presented in chapter 6 and conclusions are drawn in chapter 7.

Reference material on GPUs, as well as a list of literature, is included in the collection of appendices.

2 PC and GPU Analysis

The primary goal of this project is to get a 3D video impression out of a multiview display. Our video sequence only contains frames that describe a single point of view, as captured by a single camera. These input images need to be processed, not just streamed to the display device. This processing results in new images. The creation of such computer images or video frames is commonly called rendering.

This chapter discusses the hardware needed for real-time multiview rendering. Our focus is on personal computers equipped with programmable GPUs.

2.1 PC Architecture

2.1.1 Essential Components

A personal computer (PC) consists of three essential components, which are interconnected as is depicted in figure 2. The essential components are the central processing unit (CPU), volatile memory – also referred to as main memory or system memory – and the video card. The CPU and memory modules are placed on a printed circuit board (PCB) called mainboard or motherboard. Sometimes the video controller is also integrated on this same board.

Computer programs are executed on the CPU and perform calculations on data. Both programs and data are held by main memory. The video card enables visible output. In addition all desktop PCs generally have at least one hard disk, which serves as non-volatile memory. This is where programs and data can be stored (more) permanently.

These PC components are controlled by two additional processors, which are commonly called 'the chipset'. The chipset consists of a Northbridge chip and a Southbridge chip. The Northbridge connects the CPU, main memory and video card(s). The Southbridge masters system input and output (I/O), such as disk drives, and other system buses, for example the Peripheral Component Interconnect (PCI) bus.

Just like the CPU and system memory, the chipset is located on the motherboard. Although playback of video streams involves all of the before mentioned components, we are particularly interested in the data paths between these parts.

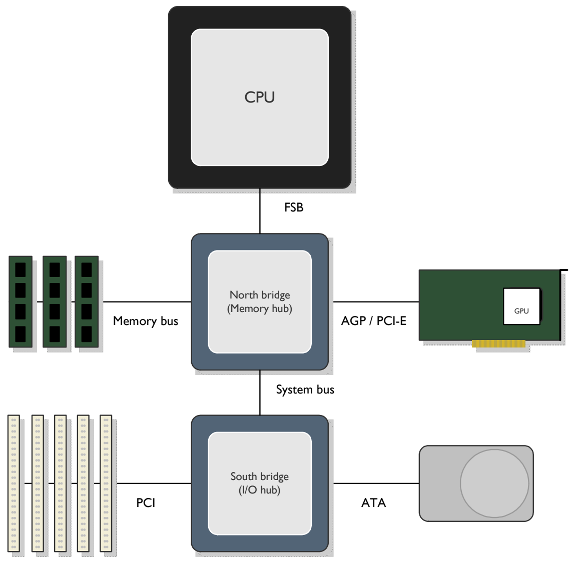

Figure 2. The architecture of a personal computer consists of functional components and communication buses.

In order to display a video stored on disk, video data must pass various buses and components. During this process video (and audio) data is continuously read from the hard disk. The data passes both South- and Northbridge on its way to main memory.

The CPU generally performs processing on this data. The executed program determines what data is moved to and from the CPU.

Finally, the buffered data is moved from main memory to the video card, which drives a display device. The application software running on the CPU controls the playback process.

Each system part must be able to sustain this relatively large data transfer. The next paragraph defines the bandwidth requirements for playback of such data streams. The subsequent sections discuss the primary tasks and limits of each PC component in detail.

2.1.2 Video Playback Requirements

The playback of video puts a relatively high strain on a computer system. Table 1 summarizes the raw throughput requirements for a few common television standards [1] in megabytes per second (MB/s). The images displayed on a television or computer monitor consist of small colored dots. Such a dot is called a pixel, which is a contraction of the words 'picture element'.

Active lines	**Width**	**Pixel count**	**FPS**	**4:4:4**	**4:4:4:4**	**4:2:0**	**4:2:0:4 (all in MB/s)**
480p (VGA)	640	307200	30	~27	~36	~14	~22
540p	720	388800	25	~28	~38	~14	~24
576p	720	414720	25	~30	~40	~15	~25
720p (HDTV)	1280	921600	25	~66	~88	~33	~55
1080p (HDTV)	1920	2073600	25	~149	~198	~75	~124

Table 1. The more pixels each video frame hold, the more bandwidth it requires for both storage and transfer. Active lines are the number of visible pixel rows. FPS denotes frames per second, which is the playback rate of motion video.

It is important to distinguish RGB data from RGBD in real-time solutions. The depth channel could be added along the path between disk and video card. It is the responsibility of a component called a depth estimator. Instead, we will be streaming pre-processed video sources that already carry depth information. Our source files use the 720 × 576p format, where 'p' denotes progressive scanning [2]. The final output of the video card is again a RGB signal without a depth channel, though the pixel count likely differs.

In practice video systems always use some form of subsampling. Technicians do not regard subsampling as a compression technique like for example MPEG-1. Over the years various MPEG standards for 'lossy' video compression have been defined by the Moving Picture Experts Group. This project does not consider these compression techniques.

Subsampling saves bandwidth. For example 4:2:2 subsampling packs the luma and chroma channels of each 2 × 2 pixel array into 8 bytes instead of 12 bytes. 4:2:0 uses only 6 bytes for each 2 × 2 array. For subsampled RGBD this results in a lower raw bandwidth requirement of

$(6 / 12 \times 3) + 1$ bytes/pixel $\times (2 \times 10^3)$ pixels $\times$ 25 FPS $\approx$ 124 MB/s.

It halves the number of bytes needed for both luma and chroma of the 2D video. The depth information still takes a single byte per pixel. Depth information can be stored in similar way as is often done for the alpha (transparency) channel. In this case the last digit of 4:2:0:4 denotes the sampling of depth in relation to luma sampling instead of alpha.

2.1.3 *Hard disk performance*

We opt for the highest possible picture quality, thus not using compression or subsampling. Neglecting the fact that we will perform multiview rendering for the moment, a hard disk has to read $(2 \times 10^3) \times 25 \times 3 \approx 149$ MB/s.

When discussing hard disk performance we need to distinguish internal from external speed. Internal speed is related to physical properties and mechanics of the drive. To be more precise: hard disks read and write using a small local memory buffer to optimize throughput by scheduling. Buffer sizes range from 1 to 8 MB nowadays.

The external speed is subject to the I/O technology used. There exist a few standards on hard disk I/O, which are ATA/ATAPI[*], SCSI[†] and serial ATA. The interface types determine the maximum throughput between the hard disk cache and system memory. The tables 2 and 3 list the maximum theoretical throughput for existing SCSI interfaces and ATA-varieties.

Name	Standard	Throughput (MB/s)
SCSI	SCSI-1	5
Fast SCSI	SCSI-2	10
(Fast &)Wide SCSI	SCSI-2	20
Ultra SCSI	SCSI-3	40
LVD SCSI	Ultra-2	80
Ultra-160 SCSI	Ultra-3	160
Ultra-320 SCSI	Ultra-320	320
Fast-320 SCSI	Ultra-640	640

Table 2. This table lists the maximum theoretical throughputs for existing SCSI interfaces.

[*] Advanced Technology Attachment Packet Interface
[†] Small Computer System Interface

Name	Standard	Throughput (MB/s)
UDMA Mode 0	ATA/ATAPI-4	16.7
UDMA Mode 1	ATA/ATAPI-4	25.0
UDMA Mode 2	ATA/ATAPI-4	33.3
UDMA Mode 3	ATA/ATAPI-5	44.4
UDMA Mode 4	ATA/ATAPI-5	66.7
UDMA Mode 5	ATA/ATAPI-6	100
UDMA Mode 6	ATA/ATAPI-6	133
SATA	SATA	150
SATA	SATA	300

Table 3. This table lists the maximum theoretical throughputs for existing ATA interfaces. The figure for UDMA Mode 6 was based on a finishing technical standard (see http://www.serialata.org/).

The first serial ATA standard, which is the successor of the (parallel) ATA standard, specifies a maximum theoretical throughput of 150 MB/s. This probably won't suffice for non-subsampled 1080p video streams. However, when settling for less video quality this throughput will do. The same holds for Ultra-160 SCSI.

When video streams are subsampled the bandwidth requirements somewhat lower and some of the picture quality is lost. With a small investment in hardware (e.g. additional drives in a RAID[*] configuration) we can overcome the problem of bottlenecking full quality streaming.

For the input stream the PAL standard was suggested. The frame dimensions of PAL video are 720 × 576 pixels. This is sufficient for multiview rendering because 9 × 720 × 576 is spread over 1920 × 1080. The purpose of this project is not to investigate how to perform real-time streaming, thus we assume it is possible to stream sequences of this quality, based on the above information. In the FPGA set-up a resolution of 720 × 540 was chosen because height of 540 pixels scales more easily to the 1080 pixels of the lenticular display device.

Our test case will be based on a set of 576p25 source videos. These videos will consume up to 30 MB/s bandwidth, which requires at least UDMA-2. This is no problem because UDMA-5 capable drives are now commonplace. A SCSI-based test system should use at least an Ultra SCSI I/O controller.

It is often suggested that hard disk performance is limited by external factors. It is true that the interface must provide sufficient bandwidth to transfer data to and from the drive. In computer sales the hard disk interface is often stated as ATA/100, ATA/133 or SATA300. These labels are not technically correct according to the specification [3]. ATA (without the number) is the name of the interface standard on parallel communication. The appended number just gives the potential buyer an indication of the maximum throughput. ATA drives are sometimes referred to as EIDE; a term once introduced by hard disk manufacturer Quantum. The specifications are correctly stated as ATA-5, ATA-6 and SATA respectively, but it gives less of a clue. In fact, the external interface does not tell us what the real transfer rates will be. Hard disk performance could as well be limited by internal factors.

For video streams it is important to have seamless playback, e.g. exactly 25 frames per second. This means we are interested in the sustained transfer rate of hard disks. However for desktop systems, which traditionally use less expensive ATA drives, such guarantees cannot be given. Most hard disk manufacturers do provide typical data transfer rates for their ATA products, as is listed in table 4[†].

[*] A redundant array of independent drives (RAID) improves performance and/or reliability.
[†] These lists are compiled from various technical papers and are available upon request.

Manufacturers	Model	Interface	Typical Transfer to/from Media
Fujitsu	MPG3204AH	ATA-5	29.6 / 50.8 MB/s
Hitachi/IBM	HDS725050KLA360	SATA	31 / 64.8 MB/s
Samsung	SP2504C	SATA	? / 121 MB/s
Seagate	ST-380023AS	SATA	27 / 44 MB/s
Western Digital	WD2500SK	SATA	≤ 93 MB/s

Table 4. This short list shows typical transfer rates for modern hard disks.

More expensive SCSI drive configurations are common in server environments. Drives, like the ones listed in table 5, often perform better and are considered to be more reliable. Because of the differences between SCSI and ATA, the former interface allows multiple drives to operate independently even when connected to the same bus. This improves performance for disk arrays, or state fully redundant array of independent disks (RAIDs). Today, also RAIDs exist based on ATA drives.

Manufacturers	Model	Interface	Sustained Data Rate
Fujitsu	MAU3147NC/NP	SCSI	147 MB/s
Hitachi/IBM	HUS151414VL3800	SCSI	93.3 MB/s
Maxtor/Quantum	Atlas 15K II	SCSI	98 MB/s
Seagate	ST936701	SCSI	63 MB/s
Western Digital	WD740GD	SATA	72 MB/s

Table 5. These best-of-class of hard disks are more likely encountered in server environments.

Most ATA-6 drives are able to stream subsampled video at SDTV resolution under optimal conditions. However, operating systems are likely to interrupt disk reading from time to time to allow other processes to access this shared resource. Uninterrupted playback also depends on a streaming-friendly file format, which will keep disk seeks low.

System characteristics such as time sharing and the sharing of hardware resources are not available to the public. This makes it difficult or even impossible to predict whether a system can prevent frame drops while playing a video. An actual PC is needed to prove that a real-time application will perform as expected. Even internal hard disk characteristics matter. By design, not all sectors of a hard disk can be read equally fast. It is wise to use some performance margin. Margins of 25% or 50% are not uncommon.

The listed hard disks read at speeds of at least 27 MB/s. When the system is dedicated to play and process video streams and the operating systems intervenes as little as possible, there's a good chance that the modern PC is able to sustain the playback at 576p (4:2:0 subsampled).

For HDTV most single disk configurations are currently insufficient. A RAID can be used to increase the data throughput. More disks can be added to the communication channel until its bandwidth is saturated. 720p requires two or three disks for both SATA and SCSI systems. Each drive must be able to transfer 17 MB of data per second. 1080p requires at least an array of three fast drives, each of which is able to sustain 25 MB/s. Most drives will be suitable, even with a reasonable performance overhead.

We can play a subsampled 576p video stream using a single drive. This is the configuration that we initially chose. We conclude that it is possible to assemble a disk array, which is able to handle broader streams. It is not our primary concern to put this to the test, as long as we know it does not limit our rendering tests.

2.1.4 *System Bus: North- and Southbridge*

The Northbridge and Southbridge make up the core logic chipset on the motherboard. The Northbridge chip typically handles communication between CPU, system memory and the video card. Whereas the Southbridge handles the system I/O.

These two chips are separated because of design and fabrication complexity. These two parts could as well be integrated in future designs. For now, the chips are linked by a high-speed interconnect. All video data from disk is passed over this communication path. On older chipsets this link formed a bottleneck, especially in application of high definition video sourced from disk or another PCI peripheral. Today's chipset bandwidth of 1 gigabyte per second (GB/s) exceeds this bandwidth requirement by far.

2.1.5 *System Memory Speeds and Bandwidth*

The set of available memory types has exploded over the last years. This makes it harder for consumers to select the right parts. Today, DRAM modules exist in various speeds denoted as for example PC2700, PC3200 for SDRAM and PC-800 for RDRAM. The latest memory technologies include double data-rate (DDR) and Rambus[*], the latter that Intel licensed in late 1996 for its motherboards. We won't go into the details of these memory types and technologies, but we are interested in its bandwidth characteristics.

At the moment of investigating the hardware, we did not know exactly what the bandwidth requirements are for depth estimation and its rendering. A precise specification of memory read and writes is needed for that. But we can estimate the order of magnitude for this requirement. Depending on the number of (sequential) processing steps on each video frame, one, two, three or perhaps ten read-writes are needed in system memory, before the data is passed on to the graphics card. Luckily, (cache) memory speeds already go up to about 35 GB/s. However, a relatively high locality or reference is assumed. It means that caches are only suitable in the situation where memory addressing is highly coherent.

Let's assume video processing perform one hundred processing steps on complete high definition video. This will consume a lean $100 \times 2 \times 2 = 400$ MB/s of bandwidth. The peak bandwidth for the older PC2100 DDR memory is 266 MHz $\times$ 8 bytes $= 2.1$ GB/s. Thus, very inefficient video processing on the CPU is unlikely to be limited by the speed of system memory. Table 6 lists other relevant memory bandwidths in different parts of computer systems.

Component	**Bandwidth**
GPU Memory Interface	35 GB/s
PCI Express Bus (×16)	8 GB/s
CPU Memory Interface (800 MHz FSB)	6.4 GB/s

Table 6. This table, taken from GPU Gems 2 by Matt Pharr et al., ch. 30, p. 472, lists typical bandwidths of PC components.

2.1.6 *CPU and the FSB*

The front side bus (FSB) is the data bus that connects the CPU to the rest of the system. The frequency of the FSB determines the external speed of the CPU. Modern motherboards feature a 166 or 200 MHz FSB and the communication is double or quad pumped. This technique exploits the both rising and falling edges of the base clock signal. The tightened timing allows for twice or four times the data rate.

[*] More on the memory interfaces of Rambus Inc. is available at http://www.rambus.com/

CPUs often use an internal clock multiplier to run the processor at the right speed. I.e. an Intel P4 660 processor runs at 4 (pumped) × 200 (FSB) × 8 (internal clock) = 3,600 MHz.

2.1.7 Video Buses

During playback a video stream is normally passed to a graphics cards once. The data again passes the memory bus twice – the second time in reversed direction – and then passes the video bus. Today, most motherboards either contain an accelerated graphics port (AGP) or PCI Express (PCIe) bus [4]. In case of an older PCI video card the data is passed back over the North- and Southbridge, instead of using the designated video link.

Before dedicated video buses were designed PCs were limited in graphics performance. Buses like VESA* local bus (VLB) and PCI could not deliver high-resolution video or complex 3D graphics without latency or frame drop. PCI is based on a 66 MHz clock. The inferior VLB was clocked at 33 MHz and 32-bit wide.

Since the Intel Pentium II generation, motherboards were designed with a single accelerated graphics port (AGP). This bus allowed faster reads from and writes to main memory, which was typically exploited to extend the video memory of graphics boards.

PCIe is the successor of the AGP. It is a new serial I/O technology, compatible with the current PCI software environment. The open industry standard exists since 2004 and is managed by the special interest group PCI-SIG†, which was formed 1992 and became non-profit in the year 2000. PCI-SIG is also responsible for the conventional PCI and PCI-X standards.

The initial PCIe documentation consisted of two parts; the base specification and the card electromechanical specification. Version 1.0 of these specifications was released in July 2002. In June 2003, after a period of enablement, compliance and PCI-SIG member reviews, the mini card electromechanical specification became available. A low-power addendum was released in September of that year to support the mobile (graphics) industry.

According to the base specification the PCIe (×16) peripheral connector should be colored black or at least a color different from PCI, which should be white. Still, the PCIe slots can be recognized because are positioned farther off the edge of the main board than PCI slots do.

In table 7 and 8, we summarize the bandwidth properties of the AGP and PCI Express interface. All speeds are expressed in MB/s.

AGP Multiplier	MHz	Theoretical Bandwidth	Actual Peaking
1×	66	266	264
2×	133	533	528
4×	266	1066	1056
8×	533	2133	2112

Table 7. Both the maximum theoretical and the actual bus speeds depend on the AGP multiplier. This multiplier determines the clock frequency of the communication bus.

* The Video Electronics Standards Association website is located at http://www.vesa.org/

† The PCI-SIG industry organization website is located at http://www.pcisig.com/

PCIe Lanes	Theoretical Bandwidth	Actual Peaking
×1	250	237
×16	4000	3800
×32	8000	7600

Table 8. PCI Express connections can use one or more lanes. This table lists the bandwidth properties for a few exemplary lanes configurations.

AGP's writing speed was much slower than its reading speed. The newer PCIe architecture features a set of communication lanes. These lanes allow transfers in both directions at similar speed simultaneously. This is the main reason why this new technology is superior to AGP.

PCIe is designed as local point-to-point connection, rather than a shared bus. Its implementation is based on the same programming concepts and communication standards as the vintage PCI. Board manufacturers need only change the physical layer of their cards to make them compatible with PCIe [5]. These devices must support single-lane (×1) links. However, to increase throughput over 237 MB/s the device must transmit its data over multiple lanes.

The PCIe standard defines ×2, ×4, ×8, ×12, ×16 and ×32 lane configurations, which provides up to 8 GB/s theoretical bandwidth. The physical bus sizes for each lane configuration differ. PCIe provides each device with a dedicated data pipeline, unlike PCI devices that shared their communication bus.

PCIe's data transmission is interleaved or what is called striped in the specification, which means that each successive byte is sent down successive lanes. This makes data synchronization more difficult on the receiving end, but increases throughput.

Like other high-speed serial interconnect systems, PCIe does have a significant protocol and processing overhead. Long continuous unidirectional transfers can approach up to 95% of its raw data-rate. The actual peaking column for PCIe bandwidth, shown in table 8, is calculated using this percentage.

We conclude this section with the remark that both AGP and PCIe suffice for our video application. It is unlikely that we need to pass back data from video memory for multiview rendering.

2.2 Video Cards

2.2.1 Overview of Internals and Connectivity

As shown in figure 3, all video cards contain the following essential components: a graphics processing unit (GPU) – sometimes referred to as video processing unit (VPU) – video memory and at least one output port, to which a display device is connected. Some video cards have an analog output such as the analog video graphics array (VGA) output. These require a digital-to-analog converter (DAC), which repeatedly reads the picture, stored in video memory and converts this data to the analog video signal.

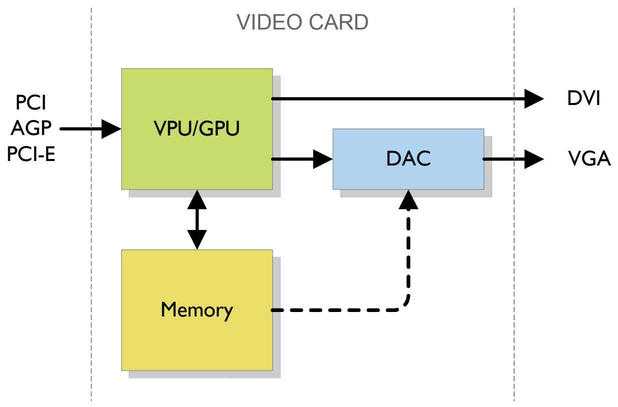

Figure 3. The video processing unit (VPU) or graphics processing unit (GPU) receives data over a communication bus. This processor interacts with onboard video memory and sends its output to DVI or VGA.

Video cards contain far more features than just writing to video memory and the subsequent 2D output to an analog or digital display device. In fact, we are particularly interested in the details of GPUs and memory of video cards. The following paragraphs give insight in what has changed on these cards over the years.

2.2.2 Graphics Hardware Evolution

Early computer graphics was limited to 2D vector graphics, where lines were drawn one after the other on the screen using coordinates. Later 2D raster graphics hardware emerged. Raster graphics allow each element of the screen (pixel) to be written to individually. Most importantly, this enabled graphical user interfaces.

In the beginning 3D applications used software rendering, which means that all render stages are mapped onto the CPU. Geometry transformations and lighting are computationally expensive. The first 3D graphics boards could handle vertex processing. Later boards accelerated rendering by means of a hardware transform and lighting (T&L) engine. Graphics processing partially moved from the CPU to the GPU [6]. On August 31, 1999 NVIDIA launched world's first programmable graphics unit, the GeForce 256. The GeForce was the first GPU to feature hardware T&L, but processing was still limited to vertex computing. ATI responded at Microsoft's WinHEC[*] in April 2000 with their Radeon chip.

New video cards brought a change to the inflexibility of the fixed-function pipe (explained in paragraph 2.2.3). In February 2001, during Intel's Developers Forum in San Jose, NVIDIA introduced the GeForce 3 for the PC platform [7]. The GeForce 3 included the same core technology as used in Microsoft's Xbox game console. It was the first programmable GPU that included a fragment processor [8]. Combined with Microsoft's shader model version 1.1 included in DirectX 8, it allowed customization of the render process by loading small programs – called shader programs – onto the graphics chip. The first programmable GPUs on the market had a separate fixed-function pipe and programmable pipe. The current trend is towards a more integrated solution.

In February 2002 the GeForce 4 was introduced, which added a new anti-aliasing technique[†], improved occlusion culling, 4:1 Z-buffer compression and better support for multiple displays.

By the end of November 2002, NVIDIA launched the FX series [9]. These GPUs series added vertex and geometry displacement support and 128-bit color precision; being 32-bits for each component of

[*] WinHEC is Microsoft's annual Windows Hardware Engineering Conference.

[†] The 2×, 4×, Quincunx™ and new 4×S anti-aliasing techniques were marketed as Accuview.

RGBA. The new GPUs were specifically designed for shader model 2.0. Next, the various components found on these video cards are discussed in order of the graphical rendering pipeline.

2.2.3 Rendering Pipeline

The rendering pipeline is a system that produces computer imagery out of 3D geometry, textures and other data. Computer graphics artists create virtual worlds called scenes. These scenes typically include 3D models, light sources and at least one camera. The calculation of the final image as seen from the camera is a complex task. It is handled by a dedicated system; the rendering pipeline. Figure 4 depicts the various stages of the classic fixed-function rendering pipeline [10].

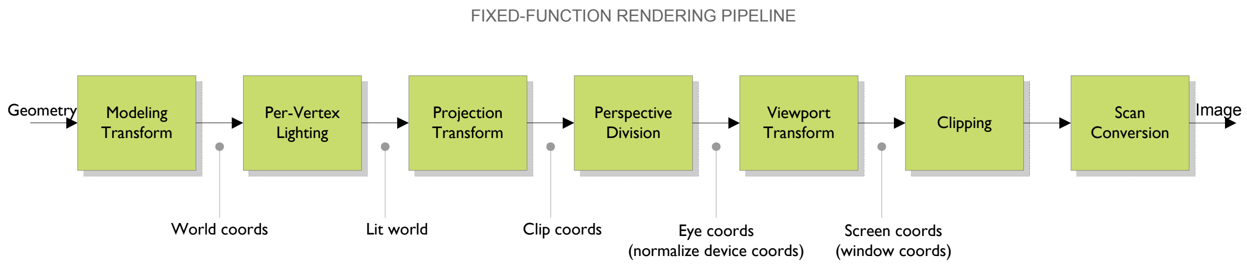

Figure 4. The fixed-function rendering pipeline consists of processing stages which convert geometry and other graphics data into output images.

In the rendering pipeline all geometry from the 3D scene is converted from model space to world space, lit, clipped and results in primitives defined by screen coordinates. Objects in 3D scenes are constructed from a set of points in space called vertices. These vertices determine the surface of the object. Most often triangles are used. These primitives interconnect exactly three points.

The modeling transform converts all vertex coordinates from modeling space to world space. For basic illumination one must at least calculate the amount of light that is incident at a triangle. This concept is called flat shading. Another solution is to determine the light at each vertex. This per-vertex lighting takes into account the amount of light received from each light source in the scene. It results in a lit world. (More information on rendering algorithms is found in chapter 3.)

The projection transform limits the 3D space to what is called the viewing frustum. The viewing frustum is a 3D volume that defines which (parts of) polygons are removed and won't be visible in the final 2D picture [11]. The removal is called clipping. Practical bounding volumes are either box-shaped or a truncated pyramid. Figure 5 shows an example of such a volume.

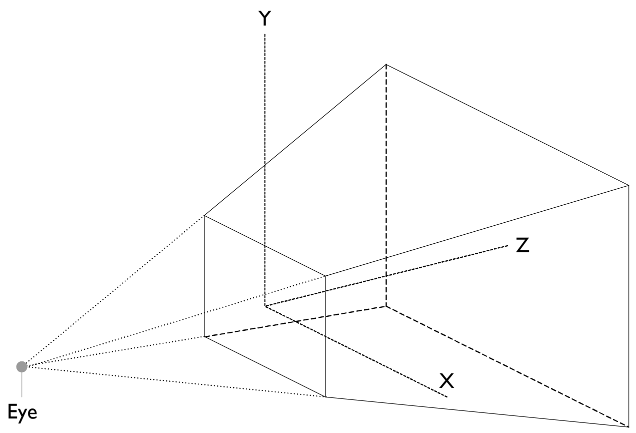

Figure 5. The viewing frustum for a perspective projection defines the visible volume of a scene. The viewing direction is along the z-axis.

The pyramid shaped volume represents the part of the scene that is visible from the camera's point of view. The six clipping planes of the volume are called 'left', 'right', 'top', 'bottom', 'hither' and 'yon'. The hither plane is the perpendicular plane closest to the eye. Yon is the farthest perpendicular plane. Often the hither plane corresponds to the viewport plane, but this need not be the case [12].

The perspective division makes objects appear smaller when they lie farther away from the viewpoint. It creates the illusion of depth. Stepping over this computational stage creates an orthogonal view. Orthogonal views have no point of view, e.g. when viewing the xy-plane from the front, there is no perspective correction along the z-axis.

The viewport transform converts eye coordinates to screen coordinates. These coordinates still have a z-component. In the next stage all vertices that lie out of the defined view port are clipped. Clipping uses linear interpolation to break off triangles at the border of the view port. The remaining triangles and other primitives are passed on to the scan converter. Figure 6 shows the stages that are part of the scan conversion process.

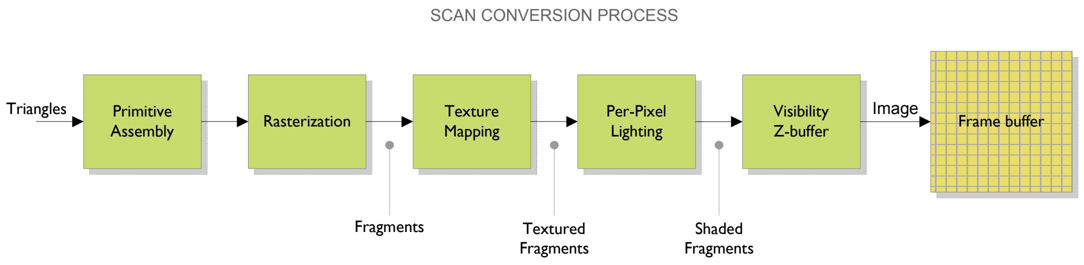

Figure 6. The stages of the scan conversion process translate triangles and other geometry into an image which is stored in the frame buffer.

The primitive assembly combines the triangles, lines and points of the scene. The rasterizer creates one of more fragments from each primitive. Such a fragment can be regarded as a pixel, but fragment is technically a more correct name as multiple fragments can make up a pixel's color. For example, a red and a green fragment can be blended to make up a yellow pixel of a particular frame.

Triangles can have an associated texture map and texture coordinates. The texture mapping stage performs a texture lookup and assigns the right color in case of such a textured fragment. More realistic illumination models take the amount of light for each pixel into account, and do not just depend on the interpolate vertex values from the rasterizer. In a typical 3D scene some parts of the object's surface could be in shade. The per-pixel lighting stage performs shading calculations for each fragment. The visibility stage uses the depth information stored in the Z-buffer to determine whether or not the fragment must be written to the frame buffer.

In the past these rendering stages were hard-coded. One could send in all data of the 3D scene and the subsequent processing was managed by the pipe itself. Post-processing the rendered images was the only enhancement it allowed. Typical post-processing effects in 3D animation include depth-of-field, glow and motion blur.

2.2.4 Programmable Architecture

Figure 7 is an example of a modern GPU architecture. It gives an overview of the components that are described next. From top to bottom, it shows the various vertex- and fragment processors, followed by the raster operators, which write the calculated pixel values to video memory.

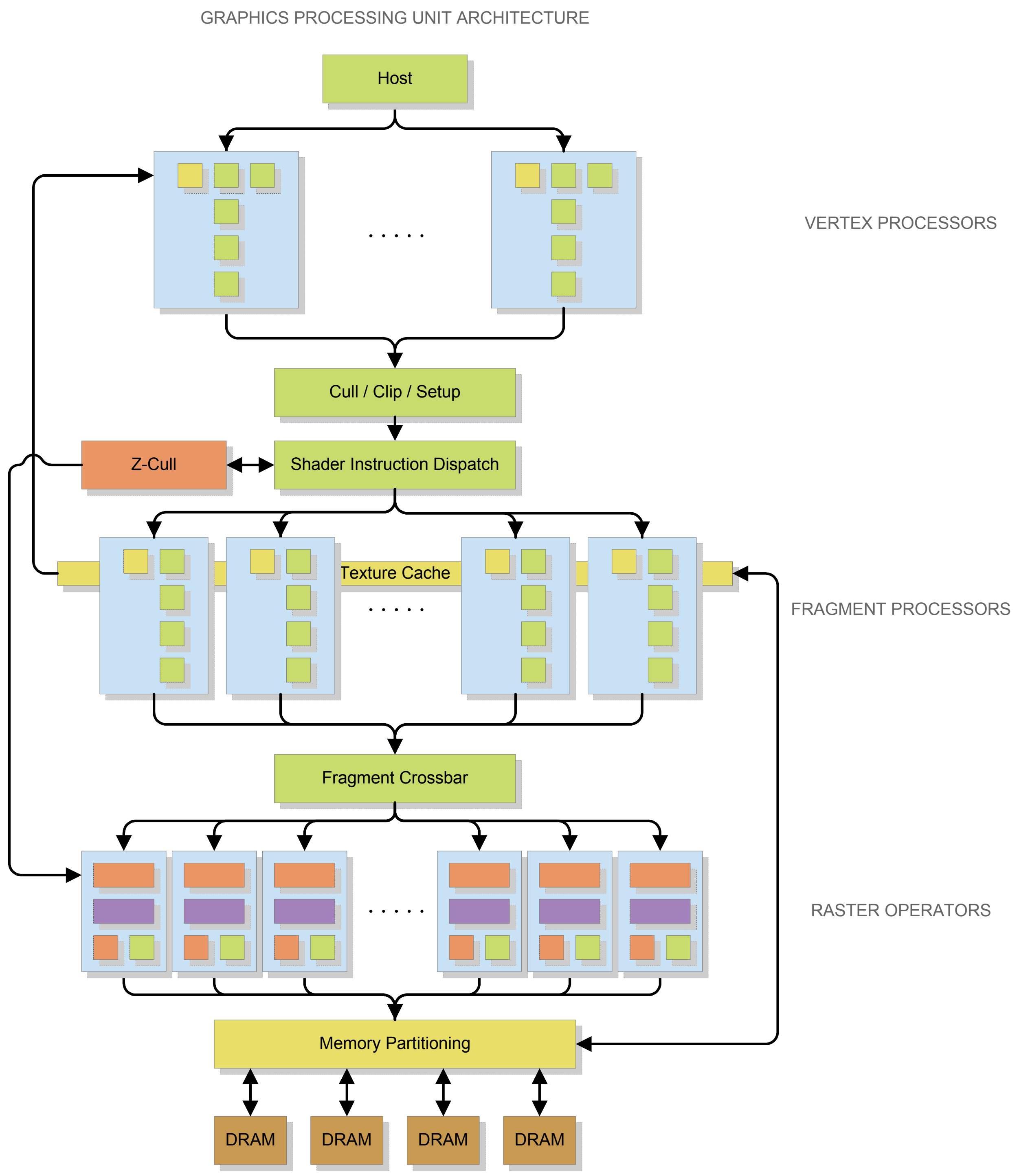

Figure 7. The processing of modern programmable GPUs can be divided in three stages being vertex processing, fragment processing and raster operations. Each stage consists of multiple pipelines that execute in parallel.

Two of the most important differences among GPUs are the number of vertex- and pixel units. Performance depends on the capabilities of these processing units and also depends to a large extent on processor timings and memory speeds. These units have capabilities conform a specific shader model, although implementations may differ among brands. Details can be found in appendices A through C[*]. Shader model requirements are listed in appendix D.

[*] Core and memory speeds are not included in the appendices because they differ among brands.

2.2.5 *Vertex Buffer*

The vertex buffer (VB) stores the geometry of the 3D scene. It holds all 3D coordinates and the index buffer (IB) determines which vertices belong to each polygon. Traditionally video cards process polygons as triangles. Triangles are flat surfaces determined by its three corner vertices. This simplifies design and increases throughput of video cards.

In case more complex geometry is sent down the graphics pipeline, the shape is broken down in appropriate triangles. This process is called tessellation.

The index buffer is used to compress vertex buffer data. As depicted in figure 8, this list holds three pointers to locations in the vertex buffer to make up a specific triangle. Without an index buffer the vertex buffer would have to contain duplicate vertex coordinates for adjacent triangles. This is undesirable, thus indexing is often used in practice [13].

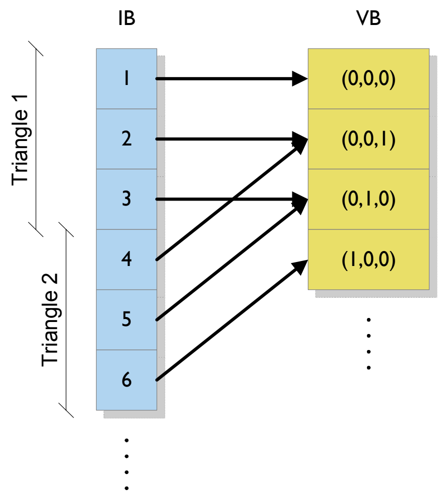

Figure 8. The index buffer (IB) lists pointers to data stored in the vertex buffer (VB).

Vertex- and index buffers can be used on all video cards that perform 3D rendering; regardless of whether it uses a fixed-function or a programmable pipe.

2.2.6 *Geometry/Vertex Processor*

The first stages of the fixed-function pipeline are part of the geometry processor (figure 9), sometimes referred to as vertex processor or vertex shader – as it computes vector coordinates and scalars.

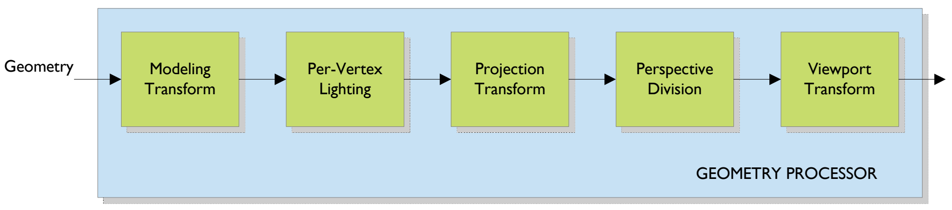

Figure 9. The rendering stages of a geometry processor handle various transformations and lighting of vertex data.

The primary task of the geometry processor is transformation and lighting (T&L). The coordinates of the input geometry are expressed in model space. These coordinates need to be converted in several steps into projected 2D screen coordinates. The lighting value at each vertex is also computed during this process. The per-vertex data will be interpolated in the rasterizer stage. The geometry processor passes its output to the scanline converter.

Modern video cards do not use scanline conversion. Instead a vertex processor (or vertex shader) passes its data to a rasterizer. The rasterizer converts all triangles into fragments (pixels). The following list shows the tasks of a vertex processor [14]:

1. Vector & scalar processing
2. Primitive assembly
3. Backface culling
4. Clipping
5. Perspective division
6. Viewport transform
7. Triangle setup

The first stage converts all coordinates from model space to world space. Not only 3D models can be sent into the rendering pipeline. Also, basic shapes such as lines and points can be inserted. The primitive assembly combines these different input entities. The following stages are strictly relevant to geometry.

Back face culling is the removal of triangles, of which the surface normal points away from the viewer and need not be rendered. Next, triangles that lie outside of the viewing frustum are clipped.

The viewport transform changes world space coordinates into viewing space coordinates. The triangle setup combines the coordinates to construct triangles according to the information in the index buffer. These geometry processor steps result in a set of the visible and lit triangles, ready to be scan-converted.

Vertex shaders as found on modern GPUs serve the same purpose as the traditional geometry processors, but allow customization of the processing. Vertex shaders can be loaded with vertex programs, which specify how transformation and coloring at vertices is done.

The minimal feature set of shader hardware is specified by a shader model. These shader models are maintained by Microsoft (see appendix D). Newer vertex processors that are 'shader model 3' [15] compliant have the ability to fetch additional data from the texture cache. This feature is called vertex texture fetch. It enables a larger set of shader constants to be loaded in the vertex processor using so-called vertex textures. The operation is considered as rather expensive, but it can be used to implement look-up tables.

Vertex programs of shaders are limited in the number of instructions. Newer GPUs support larger shader programs. Today's vertex hardware can process up to 750 million vertices per second.

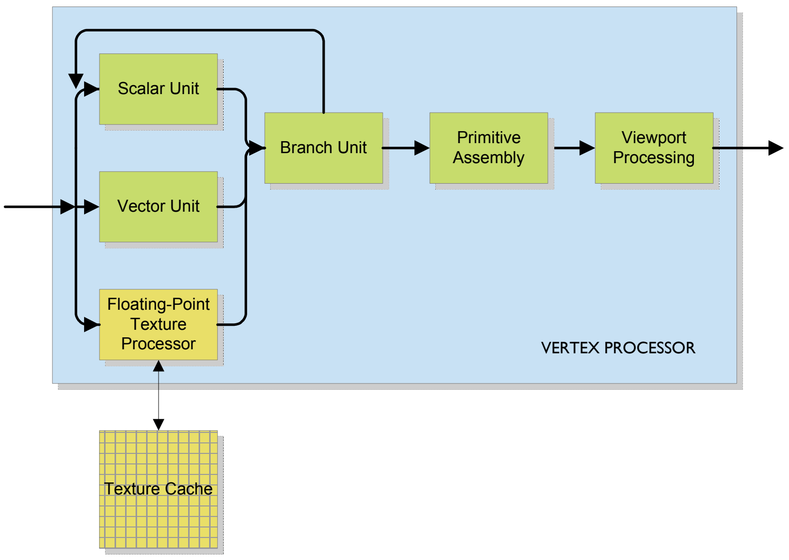

Figure 10. Modern vertex processors contain a set of dedicated computational units and have access to video memory via the texture cache. This diagram was based on GPU Gems 2 by Matt Pharr et al., ch. 30, p. 475.

The left-most units, of the vertex processor in figure 10, can perform calculus on specific data types such as scalars, vectors and texture coordinates. Selection instructions are handled by the branch unit. The computation for duplicate vertices is only performed once conform the list held by the vertex buffer. The primitive assembly reconstructs vertices into triangles according to the index buffer. Finally, all coordinates are converted to screen space including its z-component.

The total instruction count supported by GPUs varies. Minimal performance requirements and features are also specified by the shader model documentation, managed by the Microsoft Corporation. For example, GPUs conform shader model 3.0 must support at least 512 instructions. Some video cards already support 65,536 dynamic instructions. This means the hardware is able to determine whether instructions need to be executed to get the correct output. Instruction that do not affect the output will not get executed. This saves resources and time.

2.2.7 Rasterizer

The rasterizer translates triangles into pixel space. The algorithm basically loops over all triangles and determines what pixel on the raster it colors [16]. Traditionally, this raster was either the frame buffer or the back buffer in case of double buffering. On shader hardware the fragments can also be fed to a fragment pipe for further processing.

2.2.8 Shader Instruction Dispatch

On GPUs the rasterizer stage is implemented as a shader instruction dispatch. It manages the instruction decoding and scheduling of the shader at a hardware level and passes the fragments into one of the pixel shaders [17]. The shader instruction dispatch should keep all fragment shaders busy for maximal performance.

The shader instruction dispatch is also responsible for filtering the input by means of a technique named Z-culling. Z-culling excludes each invisible fragment from further processing by one of the fragment shaders. The filtered fragments are not sent further down the graphics pipe.

2.2.9 Z-Buffer

On 3D graphics hardware the Z-buffer is used to store per-pixel depth values of the projected scene. The buffer values determine if objects are occluded or visible, from the user's perspective. If occluded, pixels need not be rendered again. This greatly reduces the rendering cost of 3D scenes.

The precision of the Z-buffer is normally 16 or 24 bits. 16 bits granularity can result in visual artifacts when objects are positioned too close to one another. 24 bits provides better scene quality [18]. A stencil buffer often accompanies the Z-buffer.

2.2.10 Stencil Buffer

The stencil buffer is a non-displayable bit-plane, similar to the depth buffer. It determines which rasterized fragments must be excluded from a following rendering pass. An engineer can design tests and tag certain areas that need special rendering. This stenciling is an extra per-pixel test and a set of update operations that are closely coupled with the depth test [19].

This technique can be used for masking or rendering shadow volumes and reflections, without referring back to pre-calculated maps [20]. For example, an 8 bit stencil buffer could be exploited to render shadow projections of a scene with eight shadow casting light sources in a secondary pass. One, four and eight bits are the most common stencil buffer configurations.

2.2.11 Register-Combiner Architecture

The first programmable video cards featured new vertex processors, which could compute geometry transformations and per-vertex lighting (see paragraph 2.2.2). The rasterizer and scanline stages were left untouched. Initial fragment processing could only be performed conform a rather strict model. Later, pixel shaders evolved from this same model.

Figure 11 describes the structure of what is called the register-combiner architecture. This name stems from the two building blocks that make up the processing pipe. One is the register block. This memory unit stores the values being processed. The other is the combiner, which performs the arithmetic on these values.

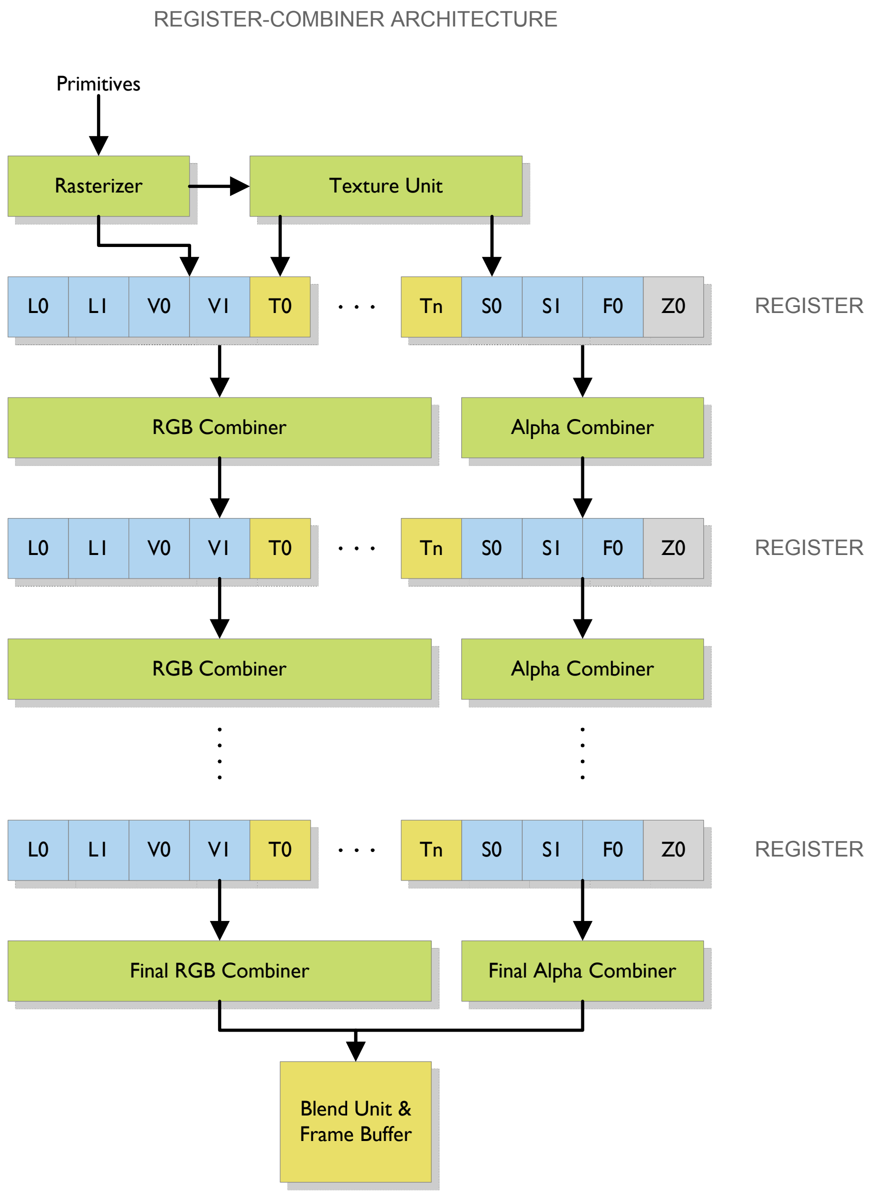

Figure 11. World's first register-combiner architecture consisted of a rasterizer, texture units followed by two pipelined register~ and combiner stages. The results could be written to the blend unit or frame buffer.

The register element can store vectors, colors and texture coordinates. Each element has an associated name based on the type of information it should hold. For example, v# are input registers, s# are texture samples and t# store texture coordinates. The number of registers available to the programmer increased each generation of GPUs. Now, these registers are exposed as variables in the various high-level shading languages.

The combiner performs mathematical operations. It reads values from several registers, operates on these values and writes the results to the next register-block [21]. Each combiner enables the computation of two dot products, or an add/multiplex operation, and a scale/bias. A combiner contains two dot/multiply units and one add/multiplex unit to accomplish this. Such a unit can be instructed to perform either one of the operations on its input, not both.

The RGB and Alpha combiners operate in parallel. The fragment pipeline consists of one or more register-combiners. The final combiner writes to the frame or back buffer. NVIDIA's first-generation GeForce design allowed two register-to-register instructions. It also featured a read-only register (Z0) and two global constant registers (C0 and C1). As of the GeForce 3 the constant registers were changed into more flexible per-stage registers (L0 and L1). The register-combiner concept was extended to a programmable fragment processor.

2.2.12 Fragment Processor

The fragment processor is also named a pixel shader. These processors can be loaded with small programs called pixel shader programs, which are executed in parallel to speed up the process. Each pixel shader receives per-fragment data like color and alpha, processes it and writes the result to the frame buffer or render target. The pixel shader has access to texture memory, but can also perform fogging, stenciling and alpha blending. Figure 12 shows three important stages of the rendering pipeline that can be customized though the use of fragment programs.

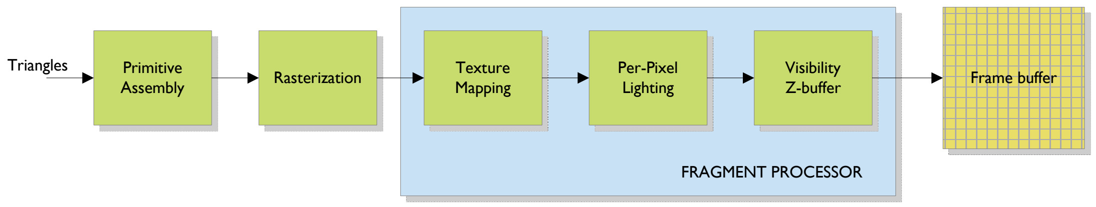

Figure 12. Pixel processing by the fragment processor encapsulates texture mapping, the per-pixel lighting and a visibility stage.

Not all fragment processors are designed the same, but they do share a few commonalities. Inside a fragment processor one or more shader units execute the instructions of the loaded fragment program. At least one shader unit can perform a texture look up from a local cache or texture memory. An (optional) texture floating-point processor handles the texture-filtering math.

As of shader model 3.0 branching operation are allowed. The branch processor handles this. The Fog ALU can blend per-pixel fog. Figure 13 shows an example of the fragment processor as can be found on NVIDIA's GeForce 6 series GPUs.

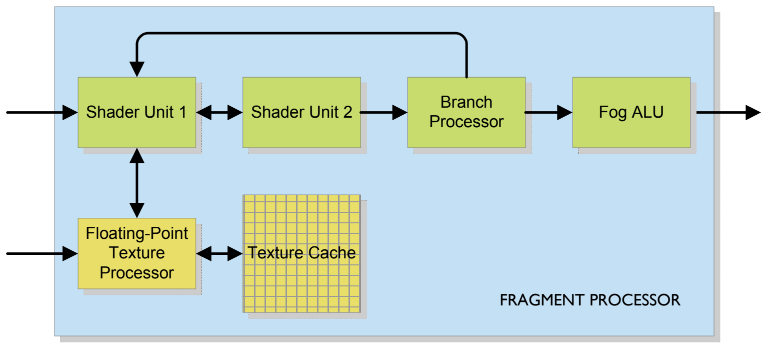

Figure 13. The fragment processor determines a fragment's color using a set of computational units as well as a texture processor which has access to video memory. This figure is based on GPU Gems 2 by Matt Pharr et al., chapter 30, page 476.

The number of fragment processors determines how many fragments can be processed simultaneously. Because of the parallel execution model, no interdependence of fragments is allowed among output values. Most cards do have multiple texture samplers, so the shader can combine multiple inputs, but such gathering algorithms are potentially slow. One of our experimental algorithms took over 100 samples (e.g. a 10 × 10 kernel) per output pixel. It proved that such an approach severely hampers performance. Similar results were obtained from our ray-tracing shader, which is described in section 4.3.

As mentioned before shader model 3.0 prescribes 512 GPU instructions as a minimum. The previous shader model (2.0) was limited to 96 instructions, being 32 texture plus 64 arithmetic instructions. More details are found in appendix D.

2.2.13 Raster Operators

The raster operators (ROPs) are part of the pixel pipeline. The fragment crossbar sits between the pixel shaders and the ROPs, so it can distribute the output of each shader to the appropriate ROP. The ROP itself performs comparison operations – like Z-culling or custom stenciling – and enables (alpha) blending and anti-aliasing of pixels in the frame buffer or render target. Some ROPs feature color and/or Z compression, like the one depicted in figure 14.

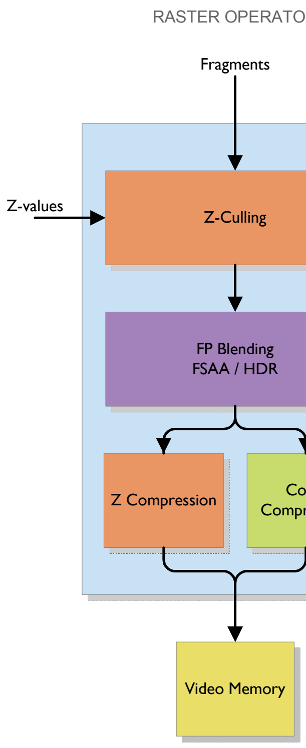

Figure 14. Modern raster operators (ROPs) perform culling, blending and anti-aliasing or high dynamic range (HDR) lighting. Some ROPs can do compression to save on video memory.

Certain video cards feature less ROPs than they have pixel shaders. This saves transistors without severely bottlenecking performance [22]. For example, NVIDIA's 6800 has 12 or 16 ROPs, depending on the specific model, whereas the 6600 has 8. The conventional GeForce 6200 has four raster operators and the TurboCache (TC) version of this card has only two.

Current GeForce 6 series video cards can do either full scene anti-aliasing (FSAA) – via multisampling – or high dynamic range (HDR) rendering, but not both. This is because both techniques depend on the floating point blending stage. Multisampling takes samples at two or more locations within each pixel and determines how much of it is covered by the primitive being drawn. If a pixel is only partially covered, the resulting color will be a weighted blend of the primitives' color and the background color.

NVIDIA's GeForce 6200 does not support color compression and Z compression. This will limit performance in cases such as 64-bit or 128-bit textures. The 6200 also lacks OpenEXR* blending and filtering [23].

*OpenEXR is a high dynamic-range image format developed by Industrial Light & Magic.

Anti-aliasing and anisotropic filtering technology on ATI cards is named SmoothVision. Anti-aliasing algorithms try to reduce jagged edges often visible in raster images. Version 2.1 of SmoothVision features gamma-corrected multi-sampling that takes up to six samples per pixel to reduce aliasing artifacts. The adaptive anisotropic filter takes up to 16 samples per pixel to sharpen blurry textures.

These filtering techniques are rather expensive and often have a large impact on performance. Most drivers provide control over filter settings to balance rendering speed against image quality.

2.2.14 Frame Buffer

The frame buffer, also called front buffer, holds the final picture data, which is displayed on screen. Current consumer displays range from 640 × 480 (VGA) up to 1920 × 1200 (WUXGA) pixels [24]. An appropriate amount of video memory needs to be allocated for the frame buffer.

2.2.15 Back Buffer

This memory also stores a final raster image, just like the frame buffer. The back buffer is useful in a technique called double buffering. Using double buffering the frame buffer holds the active image, which is visible on screen, while the new (next) image is rendered to the back buffer. When all contents are written the back buffer and frame buffer are swapped, so the new image gets exposed on screen.

2.2.16 Multiple Render Targets

Multiple Render Targets (MRTs) provide additional buffer space, to which a pixel shaders can write for storing rendered frames. Output is normally written to the frame or back buffer and is restricted to RGBA coding. The acronym RGB denotes the colors red, green and blue. The RGBA format – sometimes denotes as RGBα – represents three 8-bit colors channels and an 8-bit alpha (transparency) channel.

By using MRTs more than four 8-bit floating-point values can be kept. So, MRTs can be exploited to save on the expensive render loops of multi-pass rendering, but in practice certain restrictions apply to using this technique [25].

One should keep in mind that MRTs have a large associated frame-buffer bandwidth cost. For example, rendering to four A32R32G32B32F surfaces consumes 16 times the frame-buffer bandwidth of rendering to a single A8R8G8B8. These strings denote the format of the data structure stored in the buffer. Again the letters represent the alpha and color channels. The interspersed values indicate the amount of bits used for the coding of the alpha and/or color channels. Modern graphics boards support a specific and rather large set of data formats.

2.2.17 Texture Memory

Texture memory or texture buffer store texture maps, depth maps or other kinds of maps that determine surface properties. These texture mapping techniques are commonly used to reduce modeling complexity. Surface details are often too small and thus expensive and tedious to model explicitly.

Texture reads are relatively expensive. The processing speed of GPUs tends to increase faster than the increase in memory speeds. This also holds for writing to texture memory, but in 3D rendering reads generally occur many more times than writes. Because memory accesses potentially limit performance (graphics) programming should be memory efficient; possibly at the expense of extra processing cycles.

2.3 Brand Specific Technologies

The market leaders on video cards want to differentiate their products from the rest. The following paragraphs focus on a selection of new technologies related to video cards.

2.3.1 Memory Management

ATI's HyperMemory[*] technology exploits the PCI Express bus to extend the video memory by utilizing system memory. Because writing to system memory over PCIe is fast – unlike over AGP – this memory expansion is claimed to be without compromise of speed. All computer applications should benefit from this, especially at high display resolutions.

NVIDIA introduced a similar concept and calls it TurboCache[†]. The so-called TurboCache Manager (TCM) allows direct rendering to system memory [26].

2.3.2 Multi-GPU

Both ATI and NVIDIA provide solutions to increase performance and quality of video processing. Selected products can be coupled, but these particular set-ups do require a special main board. These boards typically feature two PCIe (×16) buses.

ATI's CrossFire[‡] workload distribution is based on a checkerboard tiling [27]. Conceptually the screen space is divided in white tiles and black tiles. All white tiles are handled by the first GPU and all black tiles are handled by the second GPU. This super tiling should result in a consistent and efficient load balancing of the GPUs.

CrossFire is designed as an open platform and the two GPU models need not be identical, but some restrictions do apply. For example, when 16-pipe CrossFire Edition card is coupled with a 12-pipe card, the number of active pipes is limited to 12 on both cards. However, individual clock-speeds remain unchanged. All applications will benefit from the dual setup. The following 'CrossFire Ready' Radeon cards can be combined with a CrossFire Edition card of the same series that holds a special co-processor:

- X850 Pro / XT / XT Platinum Edition
- X800 'Vanilla' / Pro / XL / XT / XT Platinum Edition

NVIDIA competes with their scalable link interface (SLI). It requires both GPUs to be of the same model. In Multi-GPU mode only one monitor is supported. SLI is supported on the following GeForce PCIe cards:

- 6600 'Vanilla' / LE / GT
- 6800 'Vanilla' / LE / GT / Ultra
- 7800 GT / GTX

Both ATI's and NVIDIA's implementation differ from the older 3Dfx solution, which was based on a shared PCI bus. 3Dfx increased performance by interleaving the scan lines of the output. Fill rates were improved, but geometry processing however was not. Multi-GPU configurations can also be used for striped or even more generalized processing. Such a custom design need not rely on CrossFire or SLI technology.

[*] HyperMemory is a trademark of ATI Technologies Inc.
[†] TurboCache is a trademark of NVIDIA Corp.
[‡] CrossFire is a trademark of ATI Technologies Inc.

2.3.3 Video Processing

High Definition video will become available in the near future. New optical disc formats like Blu-ray and HD-DVD and HD broadcasts will bring improved picture quality. Current PCs need some kind of hardware acceleration for smooth playback of this media. Video card manufacturers equip their new products with components to process these high definition streams. These video processors perform decoding of compressed video streams, often accompanied with image filtering.

ATI has implemented hardware acceleration for the H.264 codec and calls it VideoShader [28]. Currently, only the Radeon X700 and newer cards feature the VideoShader technology.

NVIDIA's PureVideo[*] accelerates HD video for Microsoft Windows Media files, but it requires a specific software component. It should improve picture clarity and enables high-quality scaling to any view port size. Decoding and spatial-temporal de-interlacing is supported by GeForce 6200 and higher. Hardware accelerated 3:2 or 2:2 pull down is supported as of the GeForce 6600. Certain Quadro FX cards also support this technology.

2.4 Summary

Today's personal computers are well suited for video streaming. Modern video cards with programmable GPUs enable video specialists to develop sophisticated render engines based on image processing algorithms that run in real-time.

We selected NVIDIA's GeForce 7800 GTX graphics card for developing a multiview render engine. This card was the most powerful consumer device available at that time. The 7800 is a PCI Express card. An AGP version would also have sufficed, if one existed, because we will primarily stream data to – and not from – the graphics card for our purpose.

The engine should implement the video processing in terms of vertex and fragment computations of the GPU. An elegant approach is constructing an engine similar to one for traditional 3D rendering, although multiview rendering has a few additional requirements. What these requirements are and our investigation of the related rendering algorithms is presented in the next chapter.

[*] More information on can be found at http://www.nvidia.com/page/purevideo_support.html

3 Algorithms

This chapter presents our study of rendering algorithms which were considered for the construction of our render engine. Literature provides a rather large collection of image processing algorithms, but not all of them are suitable for a GPU implementation target. We first explain the two specific problems of rendering to a multiview display. This motivates the investigation of certain mapping algorithms and 3D to 2D projections matrices, which are discussed thereafter. Then the layout of our lenticular display is explained. We end this chapter on the topic of image filtering, because all manipulation should preserve the best image quality possible.

3.1 Project Specifics

The problem of rendering RGB and depth (RGBD) to a multiview display involves two tasks. The first one is to create disparity from depth information. Once we are able to create such a set of mutually disparate views, we need to merge these views into a single image with the appropriate layout. This is the second task. The specifics of both these tasks are explained next.

3.1.1 Disparity from Depth per Pixel

To capture a true stereoscopic image one needs to use a set-up of two cameras that are positioned somewhat apart. Such a set-up is highlighted in figure 15. The resulting camera shots would include all the visual cues, including occlusion and de-occlusion. Our video sources were shot with a mono-ocular camera, but do contain depth information.

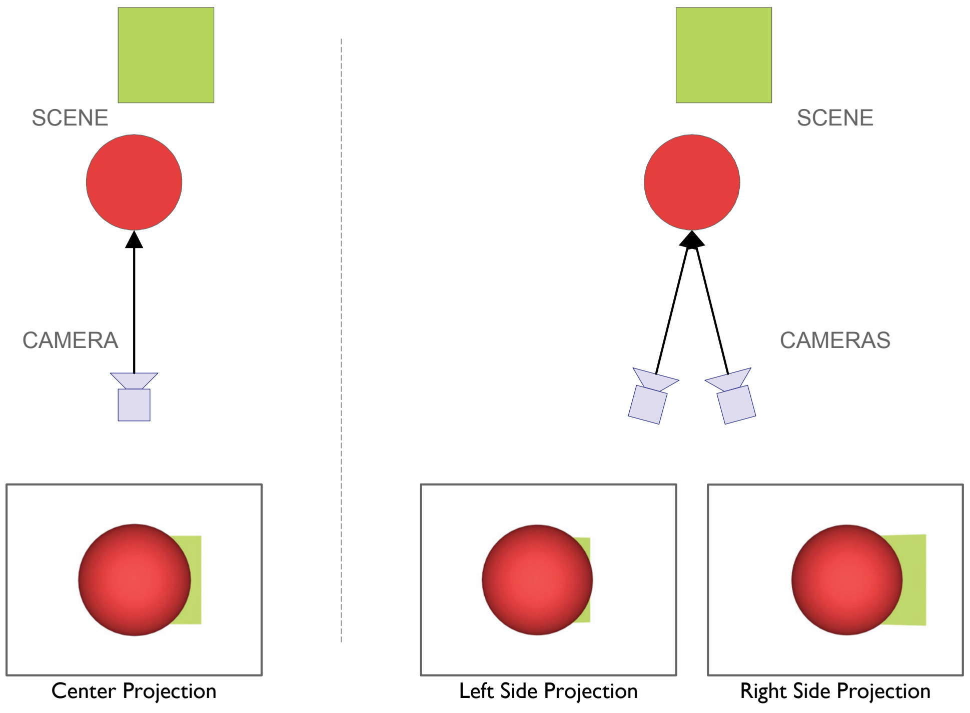

Figure 15. Stereoscopic images of (real-life) scenes can be created by using a dual camera set-up. The subtle differences of the resulting projections is what multiview rendering needs to simulate for ***N*** camera positions.

The traditional rendering pipeline for 3D graphics also uses the notion of a camera position and viewing direction. The view of a shot is controlled by one or more projection matrices. It determines what will end up in the final picture. One approach is to create disparity by changing the camera position, but we need more than that to get disparity depending on depth. This motivates our research on the subject of 3D to 2D projections together with a collection of object manipulation algorithms; topics of the upcoming sections.

Programmability of new GPUs allows us to control the rendering process. We could for example change the camera positions one or more times between output frames. Switching cameras is of little use when the scene consists of a flat texture mapped surface. To generate disparity the texture mapped object also needs some manipulation. We could either change the shape of this object or alter the texture on it. Research showed that the micropolygon-based displacement mapping algorithm translates quite nicely to shader hardware. So, the object can be manipulated using a vertex program.

3.1.2 Multiple Views per Frame

The simulation of multiple camera positions via displacement mapping results in a set of raster images. Our render engine needs to combine these images into the lenticular output format. One approach is to buffer the nine 720 × 576 RGB frames, and then multiplex the appropriate pixels by copying them to the back buffer. This is somewhat expensive in terms of video memory, because only 1920 × 1080 pixels of the pre-processed 9 × 720 × 576 pixels are effectively being used per output frame.

A more streamlined approach would not compute redundant intermediate values, but given the architecture of the GPU this is next to impossible. We cannot change the camera position in between fragments, because fragments are computed in parallel. The actual order of computation is hidden from the developer. So, we must keep the view creation and multiplexing in separate passes.

3.2 Overview

In order to create disparity we need to manipulate the input frames based on depth. This depth information typically differs per pixel, although it is spatially correlated* for objects in this picture, still we are in need of an algorithm to process certain details within one frame.

The literature on image processing provides a collection of algorithms on rendering textured surfaces – visualized in figure 16 – which proves to be of value to us. The following sections explain some of these rendering algorithms.

* We do not consider temporal coherence because of the computational complexity and associated performance penalty.

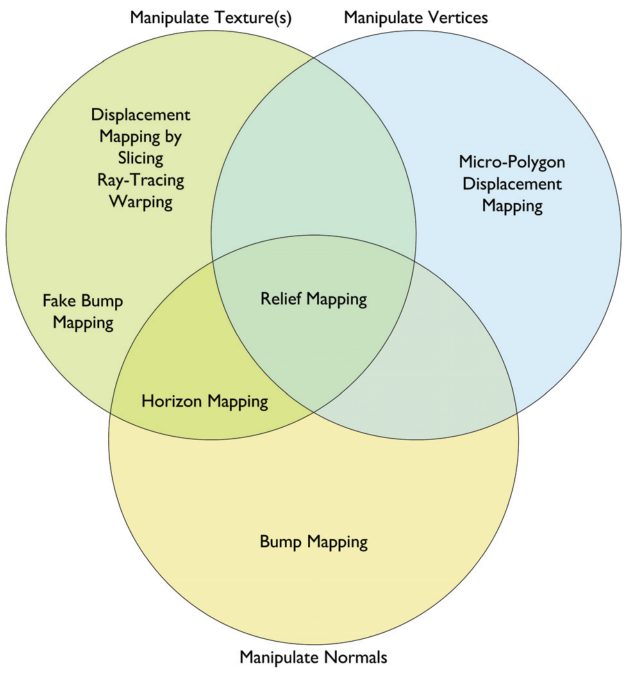

Figure 16. This Venn diagram shows the properties of a few texture mapping related techniques.

The disparity requirement rules out texture mapping on a flat surface. The derived algorithms shown in the diagram above do enable some form of sub-frame manipulation. The amount and direction of feature manipulation is either controlled by parameters such as normal vectors or vertices, or by directly manipulating the input texture. Some of the techniques combine two or more of these parameters.

3.3 Basic Algorithms

3.3.1 Texture Mapping

Texture mapping is a basic rendering technique that is used to apply a picture to the surface of a 3D object. The technique was invented by Edwin Catmull in 1974 [29]. Traditionally texture maps were used to add realism to surfaces and are still used in computer modeling and animation today. Sophisticated texturing of 3D objects could mimic real-life materials like marble or wood. Many of the more complex mapping techniques for rendering are based on texture mapping and can provide an even greater surface resemblance.

The image in the center of figure 17 is called a texture map. Such a map is coded as a 2D grid and its elements usually store color values. The texture elements (texels) can represent other data, such as height, normal vectors or other parameters [30]. The figure shows how texture mapping can be used to create a globe out of a sphere and a color map of earth.

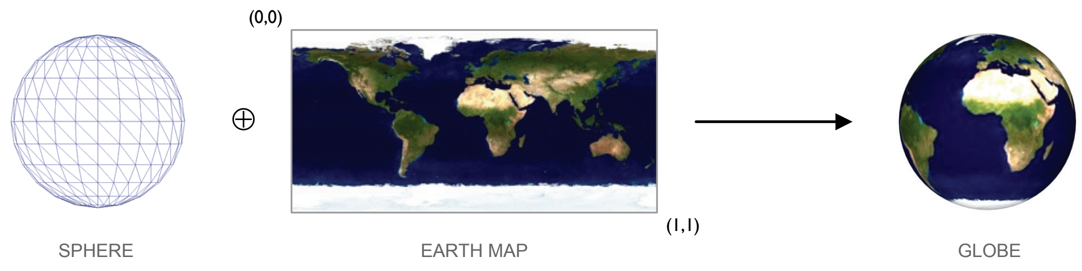

Figure 17. A globe can be rendered by texture mapping a 2D map of earth on a sphere, modeled in 3D. (0, 0) and (1, 1) are texture coordinates for the earth map.

Texture Coordinates

Texture coordinates, often denoted as (u, v), specify which part of the image ends up where on the surface of the object. The coordinates of the texture can be used to crop the edges of the texture map. The texture coordinates of the object define the position of the texture map and also enable texture tiling, where the same texture is repeated to fill a larger part of the surface.

Texture mapping is typically used to assign different colors to different spots of a certain geometric object. These objects are constructed from vertices. In case of textured objects each vertex represents a position in 3D space, but also holds texture coordinates. These texture coordinates specify where parts of the texture map end up on the object. Texture coordinates often range from (0, 0) to (1, 1) where (0, 0) corresponds to the upper-left corner of the texture map and (1, 1) to its lower-right corner. In fact, the texture coordinates pin a texture down at each vertex. When for example two vertices lie farther apart the texture is stretched between these points. Linear interpolation is often used to reduce aliasing artifacts, but picture quality could be improved using other types of filtering. Modern graphics hardware often implements bilinear or trilinear filters on texture lookups.

To use texture mapping on a model each vertex must have a texture coordinate associated. Various functions exist for texture coordinate generation. These functions determine where the texture is applied on the surface. In 3D modeling common functions are planar, cubical, cylindrical and spherical mapping. We are solely interested in a planar mapping of video frames, because the base surface is a planar grid.

Texture Clamping and Mirroring

Texture maps can be used to tile a surface. A simple example is the tiling of a bathroom floor or a brick wall in a 3D scene. This is accomplished by repeating the texture coordinates in the range [0, 1] for both u and v. Most devices allow addressing in a wider range. Addressing textures outside of the [0, 1] range can have three possible outcomes depending on the texture addressing mode, as explained by figure 18.

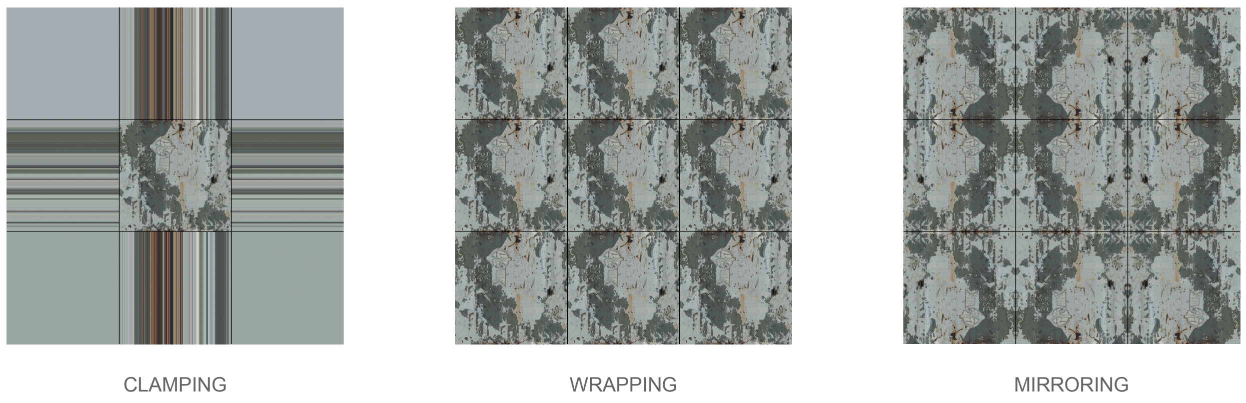

Figure 18. Common texture addressing modes are clamping, wrapping and mirroring.

The case where only the border pixel is repeated is called clamping. When the whole texture is repeated and the orientation is kept we get texture wrapping. In the last case the texture is being repeated, but the orientation is flipped. This is the mirroring texture addressing mode.

When generating views with disparity, discrepancies can occur at the vertical sides of the video frame. This is due to the shifted vertices near the border of the texture mapped grid. Clamping gives the best results, where artifacts are the least noticeable. Another more expensive approach is to prevent perturbation of vertices near the border of the grid. However, this also means no disparity is created at the sides of the frame.

Dynamic Textures

Textures can also be applied to a flat surface such as a quad. Textures need not be static. For example a video sequence can be projected onto the surface by means of a dynamic animated texture.

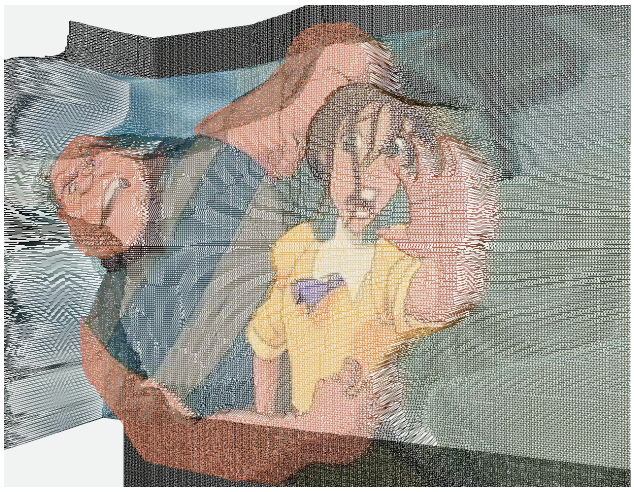

Figure 19. This texture mapped video frame shows how a fine-meshed triangle grid can be manipulated based on a suitable depth map. (source: Disney's Tarzan)

Figure 19 shows a texture map with typical texture coordinates applied to a heavily deformed surface. The camera uses a perspective projection to clarify the result. The underlying geometry is rotated about the y-axis to offer a more suggestive view. More on 3D to 2D projections is explained in section 3.5.

The level of visual detail is sometimes referred to as mesostructure [31]. Texture mapping is not sufficient for simulating mesostructure effects like (self-) shadows, occlusions, silhouettes and interreflections. Self-shadows are the result of height differences on the surface of the object, which obstructs the light from illuminating certain surface areas. Occlusion is the situation where more distant objects are (partially) hidden behind other objects, because visibility depends on the viewing direction. The silhouette – or contour – describes the shape of an object border when projected in 2D, e.g. the silhouette of a sphere is a circle. Interreflectance determines the amount of light being transferred between surface points. These points are in fact the texture elements – or texels for short.

In an attempt to simulate depth one could try to applying a texture onto a continuously changing object. Because the depth map changes each frame, the geometry updates are rather expensive. These updates cannot be easily captured by general matrix calculations as often used in 3D gaming. The depth values of the object's vertices are little related. Resending the complete object to the video card for each frame is not an option. It turns out to have too high a cost regarding bandwidth.

3.3.2 Normal Mapping

The creation of photo-realistic images largely depends on correct shading of surfaces. Various lighting models simulate how light bounces off, is absorbed or refracted by (semi-)transparent materials. One important parameter is the orientation of the surface. These orientation variations can be described by a height map. This forms the basis for a technique called bump mapping, which is elaborated on in the next section.

Another technique to describe surface variations without introducing additional polygons is normal mapping. Each texel of the a so-called normal map stores the surface normal vector explicitly. Figure 20 shows how to encode the normal map for a spike with four sides.

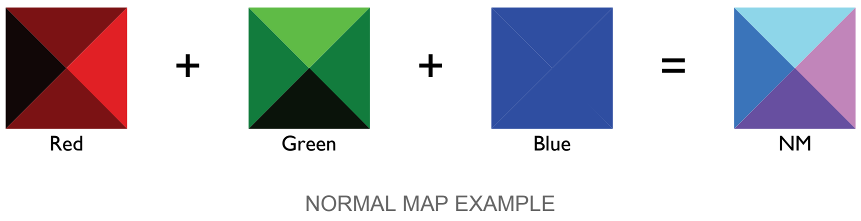

Figure 20. Adding this red, green and blue channels result in a normal map, which encodes the normal vectors of a spike.

A normal map contains 3-component vectors, rather than the single dimension of a bump mapping height map. Each vector defines to which direction a particular point is facing. The three components of these vectors – X, Y, en Z – usually are coded in the three channels of a standard RGB image. The red channel is used to encode the X direction. 100% red indicates a vector facing to the right and 0% red indicates a vector facing left. A 50% value in the red channel means the X-component of the vector is 0. The Y-component is similarly coded in the green channel, where 100% green represents a vector facing upward. The blue channel defines the Z-direction of the normal vector, where 50% blue indicates a Z-component of 0 and the vector is pointing straight out of the surface when the value is 100% blue [32]. Blue-values below 50% never occur because this would represent back faces.

Normal maps are commonly used in 3D graphics for realistic lighting of objects. We do not use lighting as part of our current render engine, but we considered using normal maps to reduce image artifacts. More on this concept is found in the section 4.5.

3.3.3 Bump Mapping

This basic technique, once introduced by Jim Blinn in 1978, is often used to add realism by shading the surface of (3D) rendered objects [33]. The existing surface normals are perturbed based on height values stored in a bump map. This subsequent illumination calculation uses the altered normals instead of the original surface normals that were derived from the geometry. The amount of the perturbation is specified by the bump map. Bump maps normally are grey-scale images. Its texel values represent height. Blinn proposed the following formula to approximate the perturbed normal n':

$$n' = n + \frac{F_u(n \times P_v) - F_v(n \times P_u)}{\|n\|}.$$

The perturbed normal is *n*' based on surface normal *n*. P_u and P_v are the partial derivatives of the surface in the *u* and *v* directions respectively. F_u and F_v are the gradients of the bump map. Other approaches exist for bump mapping. An example of this rendering technique is given in figure 21.

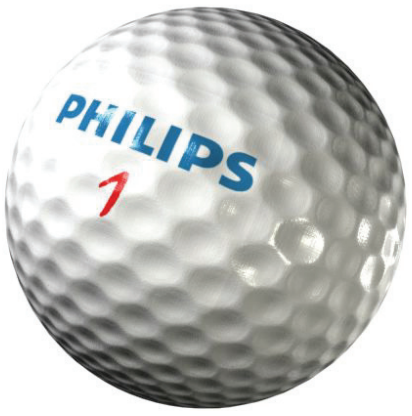

Figure 21. The bump~ and texture mapped sphere mimics a golf ball. The bump map looks like a white texture with evenly distributed black circles. The smooth gray edges of these circles result in convincing dents.

There also exists a 'fake bump mapping' technique that does not rely on surface normals. This '2D' bump mapping alters the texel indices directly. The technique assumes that all surface normals point towards the viewer, which – in general – does not hold.

Bump mapping can mimic small surface details, but depends on a lighting model. This limitation becomes evident when the 3D scene contains only environmental lighting. In this case the bumps are invisible and the surface still appears smooth; just like no bump mapping has been applied.

The algorithm neglects occlusion/de-occlusion and does not support silhouettes and interreflections either. The former problem becomes most noticeable near the sides of bump mapped objects and when the bump texture represents large height differences. In the example above the circular contour of the golf ball hints that the small dents are actually non-existent regarding the underlying geometry. This limitation is by design. Also, bump mapping depends on at least one directional light source, which renders it useless for our problem, because we do not use virtual light sources in our scene. The illumination already resides in each video frame and need not be added.

3.4 Compound Algorithms

3.4.1 Relief Mapping

Relief mapping is an extension of texture mapping where the surface elements of polygons can be displaced orthogonally [34]. It uses a simple 1D forward transform of the input texture, followed by a standard texture mapping.

The amount of displacement is defined by the heights stored in the relief map. The normal map is used for the shading features of this technique.

The algorithm computes the intersection point of a ray in the viewing direction with a height-field surface in tangent space, as shown in figure 22. It starts with a linear search to prevent from missing the first intersection. The subsequent binary search steps converge to the point of intersection more quickly. Finally, the algorithm determines whether this surface point is lit or in shade.

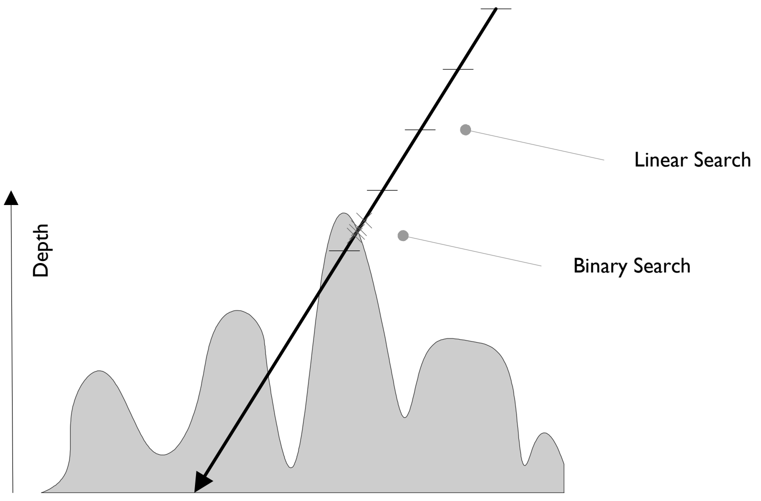

Figure 22. The height-field is intersected by a line or ray of light. The point of intersection can be computed by a series of linear search steps following by a binary search to converge more quickly. Oliveira successfully implemented this algorithm on a GPU.

In addition, the normal vectors are manipulated similar to the bump mapping technique. This results in shading – not only displacement – of the objects surface.

3.4.2 Horizon Mapping

Horizon mapping is an extension to bump mapping which includes self-shadowing of the surface. It was introduced by Max in 1988 [35]. Later Sloan and Cohen came up with a hardware implementation [36] as well did Heidrich [37]. Horizon mapping does not support occlusion and de-occlusion.

3.4.3 Displacement Mapping

The term displacement mapping represents a collection of rendering techniques. These techniques typically add more detail to polygon-based surface and support occlusion/de-occlusion and silhouettes. Several scientists each proposed a different method for displacement mapping, four of which are being discussed next. The details of bi-directional texture functions (BTFs) and view-dependent displacement mapping (VDM) are not discussed.

Micropolygons

Conventional displacement mapping, as defined by Cook in 1984, works along mesh normal vectors [38]. These normals specify the direction along which the surfaces can be altered. The so-called displacement map defines the amount of perturbation. This method was based on polygonal (mesh) subdivision, which is not suitable for programmable graphics hardware. This subdividing is often called tessellation. The REYES rendering architecture was based on a static tessellation of the base geometry into micropolygons [39]. Later a few adaptive tessellation techniques were proposed. Tessellation is not yet fully supported in graphics hardware and software implementations will not provide a real-time solution. These subdivision algorithms are inherently expensive.

Slicing

Dietrich came up with surface-aligned slices which handled occlusions correctly as long as the viewing angle was not extraordinary large [40]. Kautz extended this idea and his algorithm is based on an RGBa texture called a displacement texture [41]. This interpretation is rather similar to our RGBD frames.

Ray-Tracing

Displacement mapped polygons can also be ray-traced. Patterson proposed an algorithm which uses an inverse mapping of the rays, hence the name inverse displacement mapping [42]. His paper shows how displacement rendering can be computed for quadratic surfaces, like that of spheres and cylinders. The ray-surface intersection is calculated in parametric space instead of Euclidean space. This means there is no need for an explicit 3D construction of the geometry.

The recursive algorithm, illustrated in figure 23, works as follows. For each ray two bounding volumes are defined. The top and bottom of these volumes represent the maximal displacement of the object's surface. Each iteration these volumes are shrunk and when its coordinates, intersecting the search path, lie within a single texel this process ends. It works like the clamping of the surface with a binary search along the ray through H, M and D.

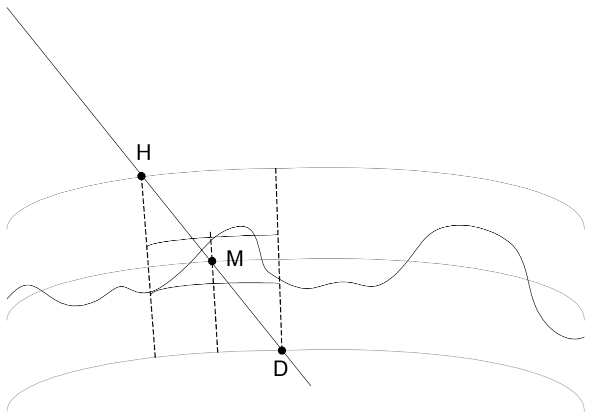

Figure 23. Ray-intersection with bounding volumes

In case of inverse mapping a planar surface, the projected rays in texture space are straight lines. Patterson's approach is of particular interest for curved surfaces, but can be generalized [43]. Given that our graphics pipeline already constructs a planar object in 3D object-space we do not benefit from a parameterized representation.

Smits' ray-tracing algorithm is based on triangle mesh geometry with vertex normals [44]. Conceptually, the normal vectors define a space partitioning into cells. We refer to figure 24. When the grid cell is entered by a ray displaced microtriangles are generated on the fly based on Barycentric coordinates [45]. The triangle pushing is along the surface normal. If the triangle is hit the intersection point can be determined, otherwise the traversal continues at the next cell. Only one tessellated cell lives per ray traversal. An explicit representation of the full geometry is prevented. This makes the method memory efficient.

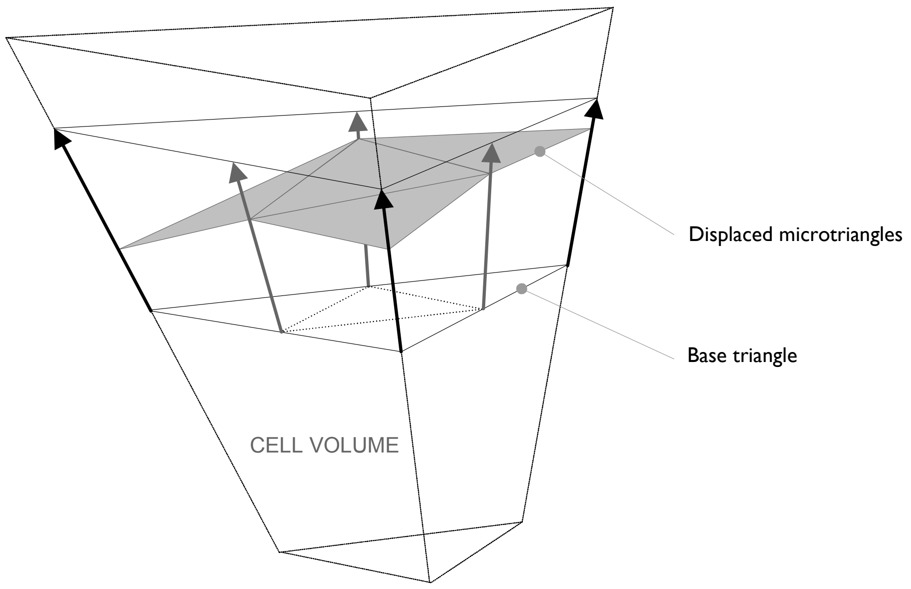

Figure 24. Each triangle defines a cell volume. Smit proposed a method that iteratively divides a triangle in four displaced microtriangles.

In case the base mesh is planar the ray tracing reduces to normal height field tracing. Musgrave showed how to render a terrain by grid tracing a surface described by a fractal noise function [46].

Warping

Displacement mapping has also been based on image warping [47]. Warping can be implemented as a forward mapping or a backward mapping. The former loops over all input pixels of the original image. The latter loops over all output pixels and searches for the corresponding pixels in the input image. This search process is rather expensive. Most warping techniques are implemented in software. One known warping algorithm for displacement mapping made it into hardware [48].

More on Displacement Mapping

Displacement mapping was adapted to support self-shadows by making it view-dependent [49]. The pixel-based VDM algorithm has been successfully implemented on programmable graphics hardware.

Displacement mapping stems from 3D rendering. When modeling solid objects, the surface normals can be generated automatically by the modeling software. However, common video input does not contain this normal information. We could try to generate a normal map from the depth map or develop an algorithm which does not depend on normals.

Research on Ray-Tracing Depth

Ray-tracing needs to sample the depth map for determining the intersection of the ray and the surface of the depth map. To guarantee the correct point of intersection every texel needs to be read from the sampler, which is rather expensive. The number of samples limits the maximum amount of disparity. This problem is caused by the requirement to sample the full range of neighboring pixels from the depth map per output (e.g. 10 to its left and 10 to its right). More information is needed to reduce this number of texture reads. One solution is to generate a distance map, which allows for a converging ray-trace instead of fixed steps (per pixel).

GPU Gems 2 describes a ray-tracing technique with uses precomputed information in order to render the displacement map from an arbitrary viewing direction. However it assumes a static depth map, whereas our wish is to support depth maps changing each video frame. Can the rather expensive precomputed be processed in real-time? This is not yet certain. The computation of the so-called Euclidean distance is a well-studied problem [50, 51], but inherently expensive. Both exact and approximating algorithms exist as well as serial and parallel implementations, but the latter do not maps well onto a GPU per se.

For the implementation of the ray-tracing algorithm for displacement mapping we choose the sample a fixed number of neighboring pixels (i.e. 10). More research on efficiently precomputed distance maps was postponed. Many publications on this topic exist, but we found little to no practical solutions, at least for GPU-compatible implementations. We currently know of only one displacement mapping example which precomputed the distance map on a CPU and it is not sufficient for real-time performance.

3.4.4 *Summary*

Up to here, we have mentioned four distinct algorithms that can alter the appearance of rendered objects. Table 9 summarizes the features of each rendering technique.

	Fine-scale Visual Effect		
	Self-Shadows	*Occlusion*	*Silhouettes*
Bump Mapping			
Horizon Mapping	♦		
Displacement Mapping		♦	♦
Relief Mapping	♦	♦	♦

Table 9. This summary table lists the supported visual features for each of the discussed mapping algorithms.

Bump mapping supports shading, but not shadowing, occlusion or silhouettes. Occlusion support is of particular interest to us, thus bump mapping is no suitable solution for out problem. Horizon mapping adds self-shadowing. Given that these shadows already reside in the video frames of the input stream, we gain nothing.

Displacement mapping does support occlusion. This technique can also render silhouettes of objects in more detail. Because our video frame will be projected on to a single quad, which borders are aligned with the view port, this last feature is of little importance.

As mentioned previously, displacement mapping can be implemented using a variety of methods. Most of these methods lack support for self-shadowing. An exception to this is view-dependent displacement mapping by Wang et al. Again, we don't specifically need shadowing. Shadows are already in the video frames. We could experiment with placing virtual light sources in our 3D scene to change shadowing, but we decided not to do so as part of this project.

Another rendering algorithm that supports occlusion is relief mapping. Relief mapping is partially based on a displacement mapping model. It also uses a form of bump mapping for the shading of objects. We consider displacement mapping to be the most suitable algorithm for multiple disparate views. Other algorithms either lack support for occlusion or have unneeded features, which undoubtedly require more processing.

3.5 Geometry Manipulation and Projections

3.5.1 *World, View and Projection*

Because our view generating algorithm is designed around the 3D graphics pipeline, a conversion step is needed to get a 2D image out of a 3D scene. Mathematician use a projection function for this kind of conversion. Such a projection can be described by one or more multiplication matrices. In case of

3D computer graphics a set of 4 × 4 matrices is used. The OpenGL graphics API specifies two matrices, being a world and a projection matrix. DirectX specifies an additional view matrix. The following paragraphs show the effect of certain matrix elements.

3.5.2 Standard Matrices

Solid objects in a virtual space are commonly modeled based on coordinates in a coordinate system. Some coordinate system to have perpendicular axis of the same scale. These are called homogeneous space and have the nice property that models do not get deformed when rotated. Other examples of spaces are affine space and projective space. In an affine space, it is possible to fix a point and coordinate axis such that every point in the space can be represented as an *n*-tuple of its coordinates [52]. Solid geometry is naturally modeled in Euclidean space (R^n), which is a type of homogeneous space.

Homogeneous coordinates allow for convenient manipulation of virtual objects in 3D space. One can change the position, orientation and size of these objects by multiplication of the appropriate coordinates. Mathematicians call these operations translation, rotation and scaling respectively. The following paragraphs describe the structure of the multiplication matrices for these standard operations.

Translate

Repositioning objects in 3D space can be accomplished by multiplying by a 4 × 4 matrix. For translation by 3-vector v the matrix has the form:

$$T(v) = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The translation matrix T shifts the coordinates of an object or scene over v_x, v_y and v_z where v_c is the component along the c-axis of vector v.

Rotate

The rotation of objects is easier to comprehend when restricting the rotation to either x-, y- or z-axis of the (local) coordinate system. Using a sequence of r_x, r_y and r_z rotational matrices is equally powerful as its product. Rotations in homogenous space can always be decomposed into a set of subsequent rotations around the x-, y- of z-axis. The following matrices define these rotations:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where θ defines the amount of rotation in radians.

Scale

The scaling of objects with respect to the origin can be done by multiplying the objects' coordinates with the following matrix:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where s_x, s_y and s_z are the scaling factors for non-uniform scaling. $s_x = s_y = s_z$ results in a uniform scaling.

3.5.3 Taxonomy of Projections

The general mathematical definition of projections is that these functions take *n*-dimensional data and map this to value of a dimension strictly lower than *n*. Computer scientists use this concept to display 3D scenes on a 2D screen. There exists more than one mapping for projecting 3D data onto a plane. Figure 25 gives an overview of the taxonomy.

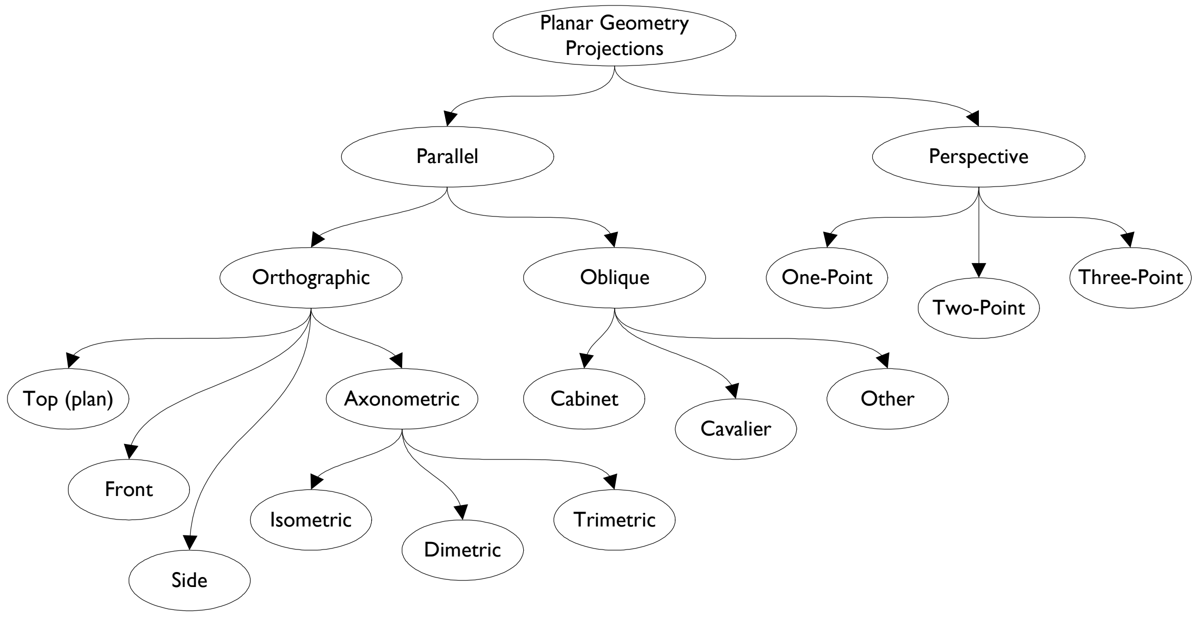

Figure 25. This taxonomy of projections shows the relation of common 3D to 2D conversions.

For parallel projections the direction of projection (DOP) is the same for all points [53]. When the DOP is perpendicular to a view plane, we can get a top, front or side view. These are all orthographic views. Cases where the DOP is not perpendicular to the view plane are called oblique, which means having an angle. Examples of these oblique projections are Cavalier (α = 45°) and Cabinet (α = 63,4°), visualized in figure 26.

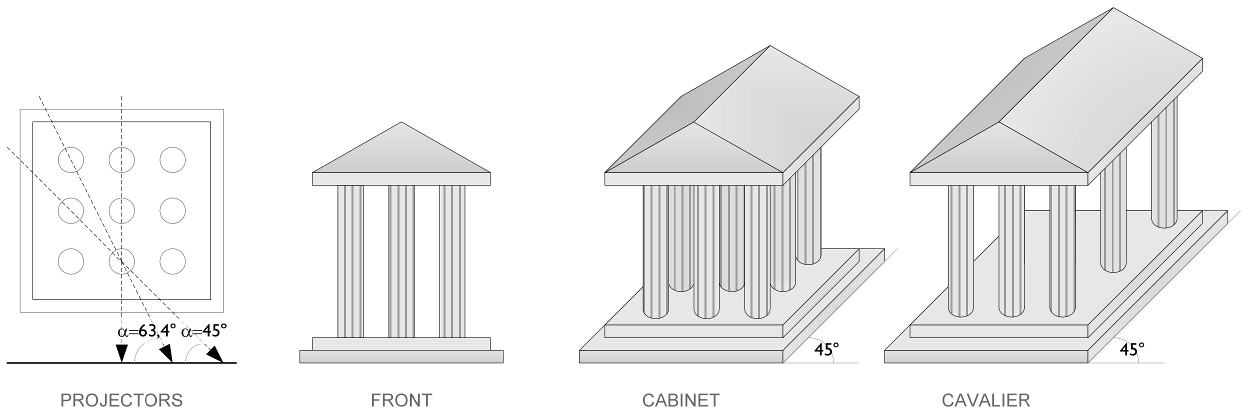

Figure 26. Changing the direction of the projectors results is three different parallel projections. The z-axis is drawn at a 45° angle for both Cabinet and Cavalier projections; this 45° drawing angle is not related to projector's angle α.

All three (axonometric) axes of an isometric projection are 120°. When only two axonometric axes are equal a dimetric results, otherwise we called the axonometric view trimetric.

Orthographic projections are true to scale but incorrect in terms of perspective. This means the size of objects appears correct, but does not vary with depth.

Unlike parallel projections, perspective projections do not have the center of projection (COP) at infinity. Points are mapped onto the view plane along non-parallel 'projectors', emanating from the COP. A perspective projection can have one, two or three vanishing points [54]. In real life there are infinitely many vanishing points, but using two-point perspective suggests nearly all the others. Next, the math behind these two projection categories will be described in more detail.

3.5.4 Perspective Projection

One class of projections is called the perspective projection. This class is not affine, because parallel lines are not preserved and rotation about x or y will distort the shape of objects. This paragraph will describe the specific properties and calculus of perspective projections.

The most prominent characteristic of the perspective projection is that it takes distance into account. This type of projection can be modeled with a knotted pyramid as depicted in figure 27. The projection plane sits in between the virtual objects and the camera position. This plane is perpendicular to the viewing direction. Objects that are more distant from the projection plane will appear smaller than objects that are positioned close to this plane. This concept is similar to what a camera would capture in real life.

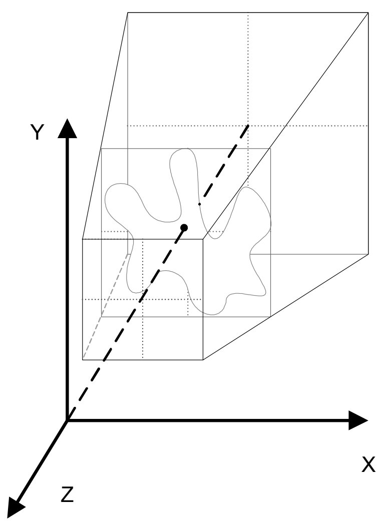

Figure 277. This is the viewing frustum for a perspective projection along the z-axis.

When using a perspective projection the virtual objects are being scaled as part of the mapping operation. This is caused by the division by distance. The viewing direction is along the w-axis by design. This explains the division by w in the projection matrix.

The transformation of coordinates (x, y, z, w) into projected – and perspective corrected – (x, y, w) can be accomplished by matrix multiplication. x and y determine the position on the 2D plane. The last component, factor w, sometimes marks the case that these coordinates are pre-transformed and thus specified in screen space. Otherwise w defines the correct scaling. Next, the generic perspective matrix is stated.

$$M_{\text{perspective}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix},$$

where *d* is the perspective correcting divisor.

3.5.5 Orthogonal Projection

Orthogonal projections neglect object to camera distance. This means no scaling occurs as part of this operation. All distance relations are respected and the transformation is affine. The projection can be visualized with a box-shaped volume, like shown in figure 28.

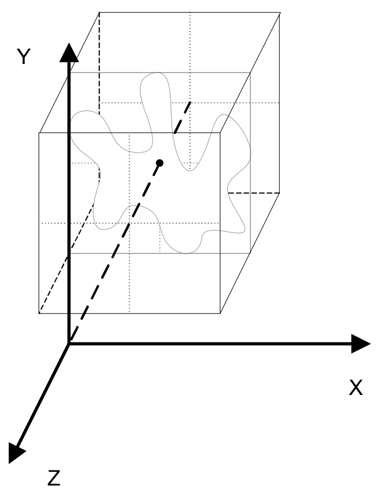

Figure 28. This shape represents the viewing frustum for an orthographic projection.

All coordinates inside this volume or frustum are mapped onto the side, which is perpendicular to the viewing direction and closest to the virtual camera. In fact, orthogonal projections can be characterized by the following. The result of the orthogonal projection only depends on the viewing direction, not the position of the camera. This direction is defined by a single three-component vector.

The multiplication matrix is solely determined by the viewing vector. In addition it is possible to define clipping planes. Only objects inside the bounded volume will be transformed. An exemplary matrix for the orthogonal projection is:

$$M_{\text{orthogonal}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This matrix drops the *z*-component to project *x* and *y* on a plane.

3.5.6 Oblique Projection

An oblique projection is constructed using the following multiplication matrix.

$$M_{\text{oblique}} = \begin{pmatrix} 1 & 0 & -\tan(\theta) & 0 \\ 0 & 1 & -\tan(\phi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where θ and φ define the angles between the viewing direction and the projection plane; along *x* and *y* respectively.

3.6 Lenticular Rendering

3.6.1 Lenticular Layout

The target display exposes a set of nine views. The surface of the screen contains small lenses [55, 56]. Without this top layer one would see all nine views at once. The lenses, however, spread light in a specific direction. A light bundle of nine views is often called a 'viewing cone' due to its shape – as is depicted in figure 30. The end-user is supposed to take position where two subsequent views within a single viewing cone are projected on to the person's left and right retina respectively. In this particular situation, it is possible to create depth vision by the exposure of two disparate views.

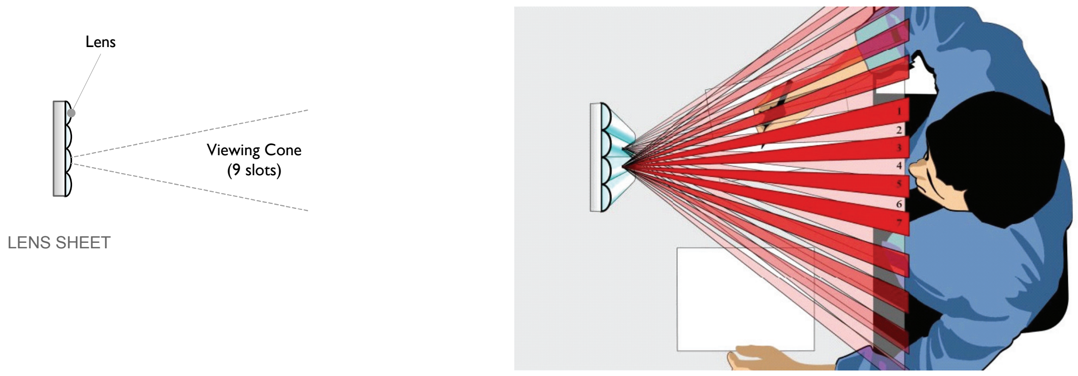

Figure 29. The viewing cone describes how the lenses spread the light horizontally with respect to the display device.

To create suitable viewing cones the lenses are aligned over the pixels in a specific way. Nine subpixels are grouped in a small patch, which can be regarded as a new screen element. These elements play the same role as the pixels of an ordinary display device. The alignment of the lenses can be explained by the tiling of the patches. To create a seamless tiling every subsequent patch is placed down two rows and one column to the right. As shown in figure 30, this offsetting results in the exact slant used for the lenses on top of these patches.

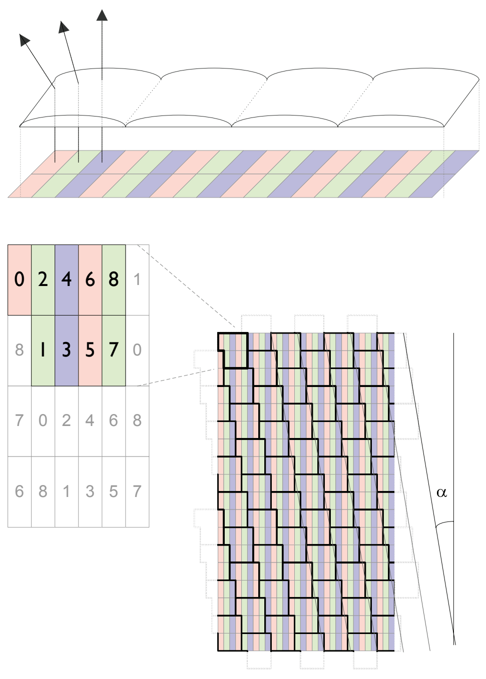

Figure 30. This diagram sketches the layout of the patches on a nine-view lenticular display. The slant (angle α) of the lenses causes a smooth transition among the views.

The colored LCD pixels underneath the lenticular sheet are aligned vertically as usual. This is because of technical and practical reasons. Current prototypes of lenticular displays are based on existing HD LCD panels.

3.6.2 Mapping Function

Shader hardware is output based, which means we must return a color value for each output pixel. Specific to our problem is that the R, G and B-value for these pixels depend on view numbers. We could compute the view for each subpixel on the fly or alternatively fetch these static values from a texture, which would trade memory for computation. In both cases we need a mapping function which takes a pixel's *UV*-coordinate and returns the view number for each subpixel:

$$f : U \times V \to N_{\mathrm{R}} \times N_{\mathrm{G}} \times N_{\mathrm{B}}.$$

N_{X} represents the view number in the range [0, 8] for the subpixel of color X. Per-subpixel formulae for our lenticular layout are:

$$f_{\mathrm{R}}(U,V) = (U \times 6 - V) \bmod 9$$
$$f_{\mathrm{G}}(U,V) = (U \times 6 + 2 - V) \bmod 9$$
$$f_{\mathrm{B}}(U,V) = (U \times 6 + 4 - V) \bmod 9$$

This mapping exactly describes the tiling depicted in figure 30.

3.7 Image Filtering

Creating a set of views with disparity from 2D video and depth information involves local stretching and compression of the input frames. These operations resample texture data and possibly introduce artifacts. Scaling artifacts, but also occlusion and de-occlusion artifacts can be reduced by using appropriate image filtering.

Image scaling is often accomplished through convolution of image samples with a single kernel. For example, we can perform scaling using a bilinear or bicubic kernel [57], but more complex methods exist. A kernel defines the set of samples being considered and the amount each sample contributes to the resulting pixel. Kernels can be regarded as a mathematical description of an image filter. The sampler component of the texture unit in a GPU already implements the description for bilinear filtering. Custom filters can also be written for GPUs.

3.7.1 Native GPU Filtering

The texture samplers of current GPUs only support a limited set of image filters in hardware. The following paragraph describes the structure of modern texture samplers as is stated in Microsoft's DirectX SDK. The subsequent paragraphs elaborate on the concepts of basic sampling techniques.

Minify, Magnify & MIP

Programmable GPUs are required to distinguish three sampler states. These sampler states define the type of filtering being used when fetching texture data. The states are texture minify (`MINFILTER`); magnify (`MAGFILTER`) and so-called mip mapping (`MIPFILTER`). Each state can be associated with a specific type of filtering or no filtering at all. The latter technique is called nearest-point sampling. The native filtering types are described thereafter.

Nearest-Point Sampling

Nearest-point sampling – or nearest neighbor sampling as it is often called – is the least expensive means for fetching data from a texture map. This sampling algorithm interpolates the texture by convolving it with a box function. A convolution is the operation of a mathematical function on a signal and is often used to describe filters; also in digital imaging.

In mathematics, the convolution theorem states that under suitable conditions the Fourier transform of a convolution is the point-wise product of Fourier transforms. In other words, convolution in one domain (here: space domain) equals point-wise multiplication in the other domain (here: frequency domain). Versions of the convolution theorem are true for various Fourier-related transforms [58].

Operationally the system selects a single texel from the map based on the specified *U*- and *V*-coordinate. The sampled texel coordinate is guaranteed to be the closest to (*U*, *V*). Because of possible ties in the selection at texel boundaries problems arise.

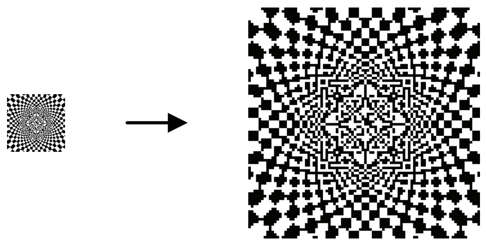

Figure 31. Nearest point upscaling will result in aliasing.

Like figure 31 shows, scaling textures using nearest-point sampling will introduce aliasing artifacts, which are perceived as 'blockiness'. This sampling technique is the least performing scaling algorithm regarding picture quality. It should only be considered for speed.

Linear Texture Filtering

Linear interpolation algorithms are typified by a triangular convolution function. The one-dimensional sampling of the linear function weighs the two nearest points. Bilinear texture filtering applies the concept of linear interpolation to both horizontal and vertical dimension of a texture. This technique first determines the texel closest to (*U*, *V*). Then it computes the weighted average from a total of four texels based on three interpolations, as figure 32 explains.

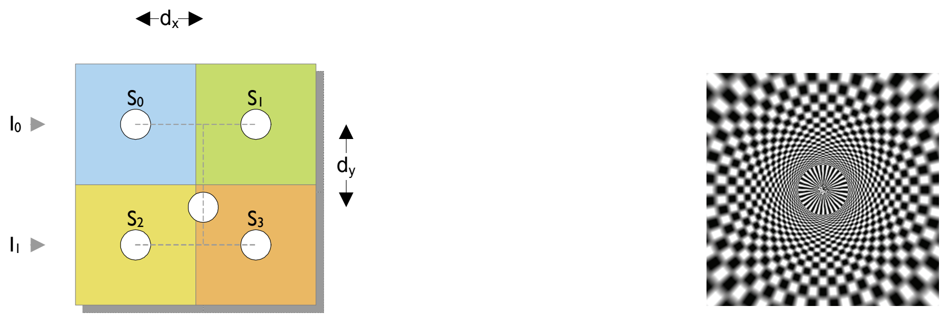

Figure 32. Bilinear filtering takes four samples for each output value and gives a smooth result.

The following formulae specify the computation of a bilinear texture lookup.

$$I_0 = S_0 \times (1 - d_x) + S_1 \times d_x$$
$$I_1 = S_2 \times (1 - d_x) + S_3 \times d_x$$
$$B = I_0 \times (1 - d_y) + I_1 \times d_y$$

The first equation linearly interpolates the samples S_0 and S_1 based on distance d_x. The second equation does the same for S_2 and S_3. The final step interpolates the previous results I_0 and I_1 based on d_y.

Bilinear filtering reduces aliasing, but could still produce jagged edges. It has a relatively low overhead and should have no impact on performance, whereas modern graphics hardware is designed to support at least this type of filtering. It is suitable for modest scaling of textures applied to surfaces that are perpendicular to the viewing direction. Angled surfaces need anisotropic filtering, which is discussed next.

Anisotropic Filtering
When a texture is applied to a surface that has an angle with respect to the plane of the screen certain distortion occurs. The more such an unfiltered surface is sloped into distance the fuzzier it will look. The artifacts are caused by the depth depended scaling of the texture map (in case of perspective projections).

A single output pixel could combine information from a number of horizontally and possibly different number of vertically spaced texels. The problem requires a non-square filtering pattern – hence the name anisotropic – that differs along the x- and y-axis. Figure 33 shows a pattern, both with and without anisotropic filtering.

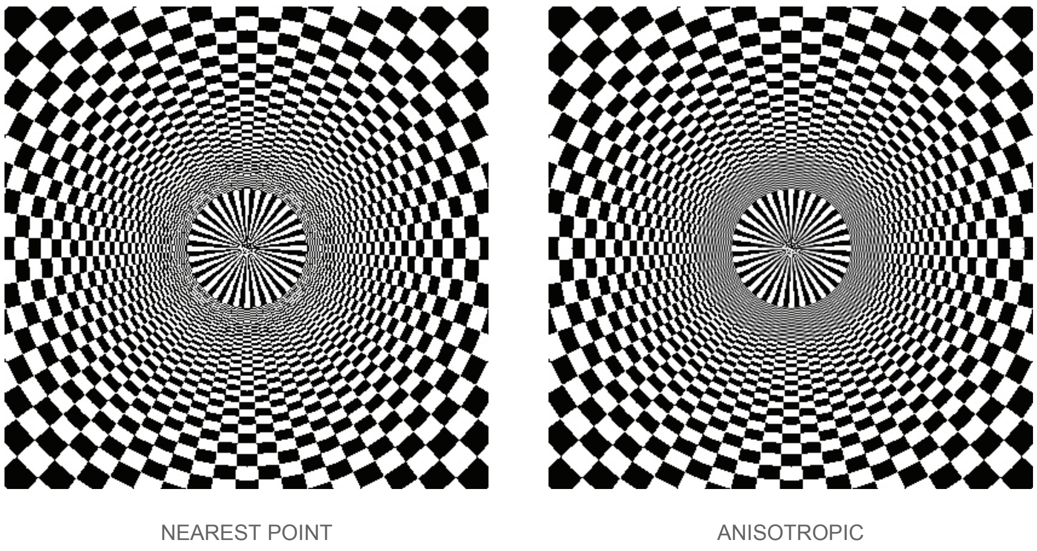

Figure 33. This nearest point sampling versus anisotropic filtering comparison shows differences near the center of the image. The image was constructed by applying a checkerboard pattern to the inside of a tube. It demonstrates the quality of both magnifying and minifying the texture.

This type of filtering is computationally expensive. Classical anisotropic filtering uses sixteen samples, called taps in the filter jargon, for each output pixel. These samples are preferably already trilinear (which takes four samples from each mip level) or at least bilinear filtered. The often adaptive implementations and resulting image quality differs per hardware manufacturer [59].

When the anisotropic filtering is activated via ATI's driver, as it usually is, the impact is less important as the tri-linear between mip-map is only processed on the level 1 texture with Direct3D. The others mip levels remains in bilinear filtered [60]. This means that anisotropic filtering is only enabled in specific areas of the output. Our video processing does not need anisotropic filtering, because our base geometry is orientated perpendicular to the viewing direction and manipulation is only modest. More so, we decided not to rely on anisotropic filtering given uncertainties as the one state above.

Mip Mapping

Mip mapping is a technique that stores multiple downscaled and filtered replicas of a high-resolution texture in a single space. MIP stands for *multum in parvo*. The phrase is Latin and translates to "many things in a small space". An exemplary mip map is shown in figure 34. Using these pre-filtered textures lowers processing time and reduces moiré artifacts.

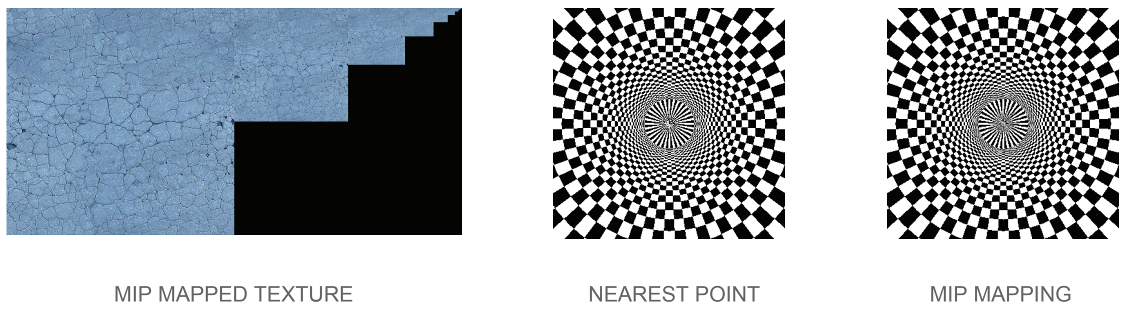

Figure 34. A mip map contains one or more pre-filtered versions of the base texture and this gives better results than nearest point sampling does when downscaling. The differences are just about visible near the center of these two examples.

3.7.2 *Custom Filtering*

Bicubic Filtering

Bicubic image filtering uses sixteen samples per output pixel [61]. These samples are formatted in a 4 × 4 cell grid, as figure 35 shows.

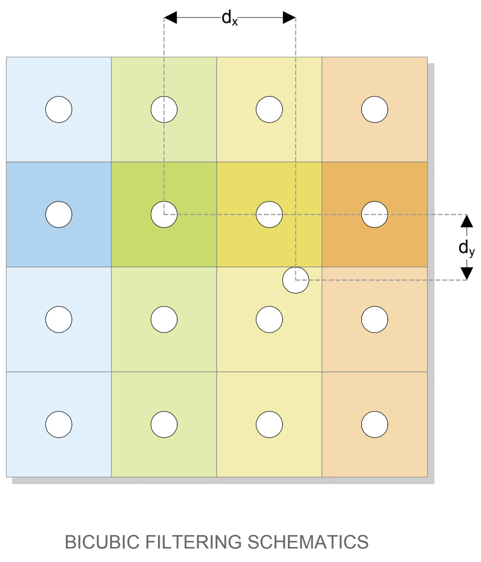

Figure 35. The concept of bicubic filtering is to combine 16 samples; weighted by distances d_x and d_y.

The bicubic filtering algorithm computes a weighted average from a cubic function in both horizontal and vertical direction. Here, 'bi' means 'in two directions'. Bicubic interpolation can be described as follows:

$$\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$$

For the sake of simplicity we first explain one-dimensional (1D) cubic interpolation. There are two common methods for averaging the samples based on cubic interpolation. The standard approach uses a cubic B-spline interpolation, the other approach is a more general cubic interpolation function, which allows for better high-frequency performance and this can be controlled by a free constant. The B-spline is in its turn a generalization of the Bézier curve. These curves are constructed by Bernstein polynomials. We will not describe this construction here. A cubic B-spline can be written as:

$$f(x) = \frac{1}{2}x^3 - x^2 + \frac{4}{6} \qquad x:(0,1)$$
$$f(x) = -\frac{1}{6}x^3 + x^2 - 2x + \frac{8}{6} \quad x:(1,2)$$

A B-spline is piece-wise continuous by definition. In this case it is defined over the interval (-2, 2) and is symmetrical around $x = 0$. For any offset the sum of the sampled interpolation function points is equal to 1 [62]. The sampled points are computed by $f(d), f(d-1), f(1+d)$ and $f(2-d)$ where d is the offset from the nearest pixel. Figure 36 is an example of cubic B-spline sampling.

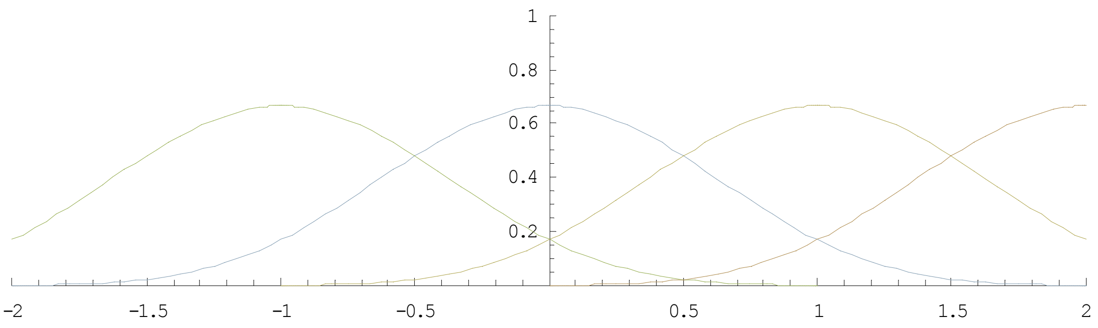

Figure 36. This plot shows four cubic B-splines which sum to 1.

The B-spline approach for bicubic filtering uses this four point sampling in both horizontal and vertical dimension. In the two-dimensional case offset d is split up in d_x and d_y.

Cubic B-splines are a special type of cubic splines. The cubic polynomials for the general case have eight unknown coefficients. For a smooth averaging of the samples we require a symmetrical and continuous interpolation function. This lowers the number of undefined – free to choose – coefficients. We also require $f(0) = 1$, $f(1) = 0$ and $f(2) = 0$. This set of constraints defines the cubic spline up to a constant (a). Mathematically, the generic cubic spline interpolating function can be written as:

$$f(x) = (a+2)x^3 - (a+3)x^2 + 1 \quad x:(0,1)$$
$$f(x) = ax^3 - 5ax^2 + 8ax - 4a \qquad x:(1,2)$$

Common choices for the constant are $a=-1$, $a=-\frac{3}{4}$ and $a=-\frac{1}{2}$. Figure 37 shows a graph for $a=-\frac{1}{2}$.

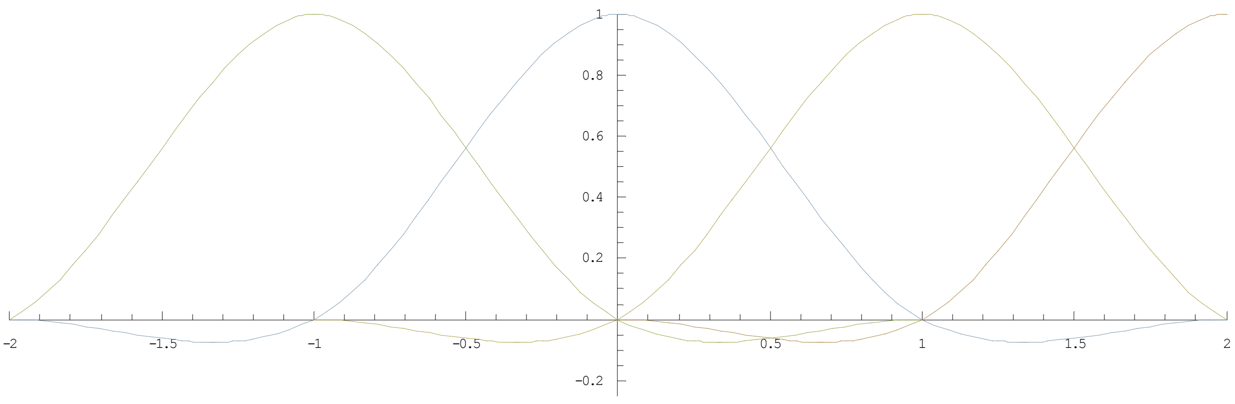

Figure 37. This plot shows four general cubic splines using lobe constant $a=-\frac{1}{2}$.

Either approach for bicubic filtering is computationally more expensive than bilinear filtering, but it usually leads to shaper image stretches.

Multi-tap Filtering

Modern pixel shader hardware does not limit us to traditional filtering techniques such as bilinear filtering. As long as the memory bandwidth suffices, we could write a custom bicubic filter or even a multi-tap filter. The downside is that the graphics hardware does require an output based implementation and multiple taps will probably require a lot of texture fetches.

Design Notes

The creation of disparate views uses subtle image manipulation, but only in horizontal direction. The amount of shifting in the color map is based on information from the depth map. Both sources must be sampled and resized, but only by a small scaling factor (e.g. ≤ 1.1). This means that picture quality will not benefit much from techniques such as mip mapping. Bilinear and bicubic filtering however will improve the quality of the output, especially near borders (in the depth map). See figure 38 for a comparison of these two filter techniques.

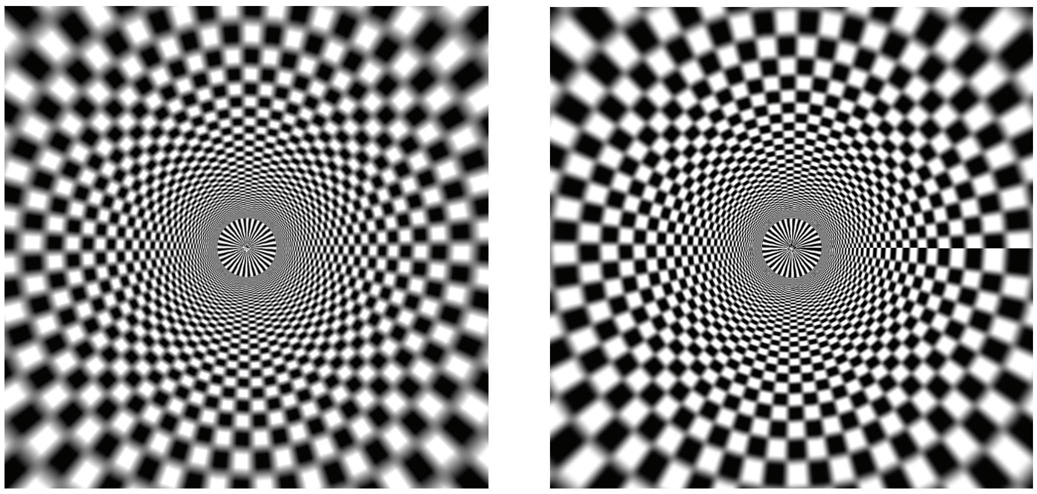

Figure 38. This bilinear versus bicubic filtering comparison shows that the latter method results in a somewhat shaper image. This is visible especially near the border.

Native bilinear filtering or a custom bicubic filtering partially wastes processing power, because vertical scaling will not occur or could be prevented by design. From a performance point of view a multi-tap filter that only samples in horizontal direction would be a better solution. This is only appropriate in the case there is room left in the pixel shader pipes for this filtering computation.

We can design a suitable multi-tap filter. What is regarded as suitable depends on the purpose of this filter and also the input characteristics. We assume only modest scaling in horizontal dimension is needed. We do not know the actual contents of the input video frames a priori. The frequency profile of images is generally not known in advance and must be treated as being random. Also, the characteristics of the accompanying depth information is yet unknown. All this makes it hard to design a good multi-tap filter.

3.8 Summary

Micropolygon displacement mapping is a suitable candidate for implementation on a GPU, because texture lookup and vertex manipulation belong to the set of shader operation. Also, modern GPUs are capable of handling a large number of polygons. The concept of displaced vertices and projecting the constructed geometry orthogonally will result in our disparate views. A separate shader program is designed to merge these views into a single lenticular output.

We refrained from using anisotropic filtering, because horizontal disparity only requires scaling in that single dimension. We chose for native bilinear filtering for performance reasons.

4 Concepts and Design

4.1 Introduction

This section describes a series of concepts that led to the current design of the rendering engine. Understanding the programmable features of a GPU need not result in a good design with respect to performance. We will discuss a few intuitive approaches and why they do not guarantee high frame rates. After this introduction the details of the candidate engine are explored.

4.1.1 Performance Considerations

As previously mentioned, the two key aspects of the engine is to create a set views with disparity and merge these conform the lenticular layout. Because of the real-time performance requirement we must use all the processing power available in the GPU. It is difficult to see what the penalties are for certain design decisions beforehand. We performed research to find out which shader operations are inherently more expensive than others.

We designed an engine that is similar to a standard 3D rendering engine – just like the ones used in computer games. This reduces the chance of abusing the hardware in a way that will probably slow performance. It is best to use a single effect with as little render passes as possible. An effect – or shader – consists of a vertex program and a fragment program, both which run on the GPU. Effects are assigned to 3D objects and define its appearance, e.g. color, reflectivity, transparency, etc. More information can be found in chapter 2.

4.1.2 Multiple Views per Frame

Traditional 3D game engines only render a single view at a time. Other important observations are that only a few scene parameters change between frames, which are for example the camera position and viewing direction and only a handful (counting polygons) of dynamic objects. The view is described by two or three multiplication matrices. Moving or deforming objects require geometry updates, which are fairly expensive regarding upload bandwidth to the graphics card. We can say that the biggest part of the scene data is static, including textures and most of the geometry. This notion has influenced our design to a large extent. The fastest way is to deform objects in video memory by using a vertex program.

We need to solve the problem of generating a set of nine views instead of only one per frame. Normally each render pass is only allowed to write its output to a single 2D buffer. This would mean that preparing all of the views would cost eight passes, assuming the center view is the unmodified RGB frame.

The introduction of multiple render targets (MTRs) loosened this single buffer restriction. The current GPUs only support up to four additional buffers. Exploiting this buffer space has its drawbacks. Render targets must be of certain texture format and are all required of having the same dimensions. Also, we only have four of them, where we logically need eight. This is another problem of rendering multiple views. The MRT-feature was designed to enable writing to multiple buffers within a single pass, which also means a single point of view.

So far we have described one of the main problems, when using hardware designed for accelerating standard 3D rendering for multiview rendering. Besides the problem of multiple views, we must also manipulate the contents to create disparity.

4.1.3 Creating Disparity with Occlusions

Chapter 2 showed that image manipulation can be the result of direct texture manipulation, geometry modification or even a combination of both. Texture mapping is a basic but also fast operation on modern graphics hardware, which we wish to exploit. As of today, relief mapping and a form of displacement mapping have been successfully implemented on GPUs. Two side notes apply. First, relief mapping supports lighting; a feature we do not need. Not exploiting this feature, the technique boils down to a type of displacement mapping. And secondly, acceptable frame rates for ray-traced displacement mapping only hold in case of a static depth map. Again, this is not a suitable choice for multiview rendering of video sequences.

We decided to design our render engine around micropolygon displacement mapping. A modern GPU can process millions of textured triangles per second. A 2D grid in 3D space will serve as a drawing canvas. The grid itself consists of a single triangle strip. The vertices of this grid not only specify the position in 3D space, but also the texture coordinates needed for texture mapping.

Changing texture coordinates would only result in stretching and compression of local image features. This is because the rasterizer stage of the GPU linearly interpolates this vertex data. This technique proves to be a basic solution to the disparity problem, where pixels can be shifted horizontally. The downside is that features never become occluded. Figure 39 shows how an orthogonal projection of a 3D shape can result in horizontally shifted pixels.

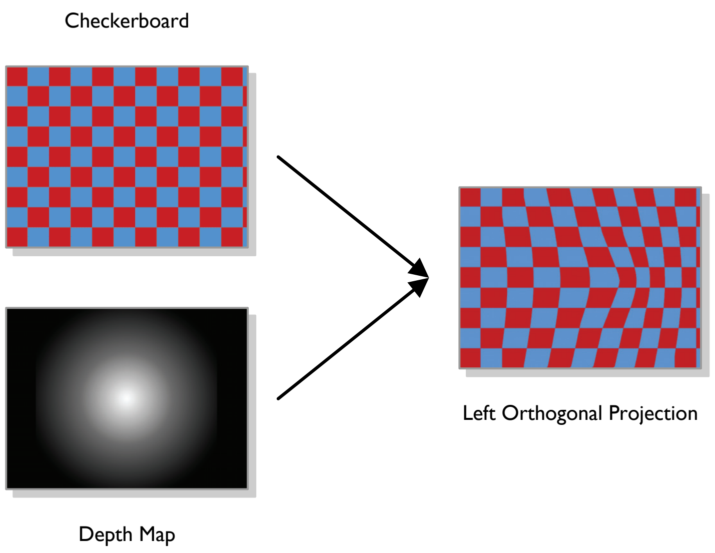

Figure 39. The horizontal shifting of a checkerboard pattern is the result of manipulating the underlying geometry by the amount specified in the depth map.

Instead of manipulating the texture coordinates, we manipulate the z-value for each vertex depending on a value stored in the depth map. The shape, resulting from displacing coordinates of the triangle grid along the z-axis, represents the depth of the real-life scene. A texture map is applied to the surface of this new shape. Using a specially constructed orthogonal projection we can create occlusions. We have not solved the de-occlusion problem yet.

One important property is that the resulting amount of disparity after projection is linearly related to the amount of shifting of vertices along the z-axis. We will exploit this property by rendering multiple views simultaneously, simulating the camera position shifted either to the left or to the right side. An example of a manipulated triangle grid is given in figure 40. Details are described in section 4.1.4.

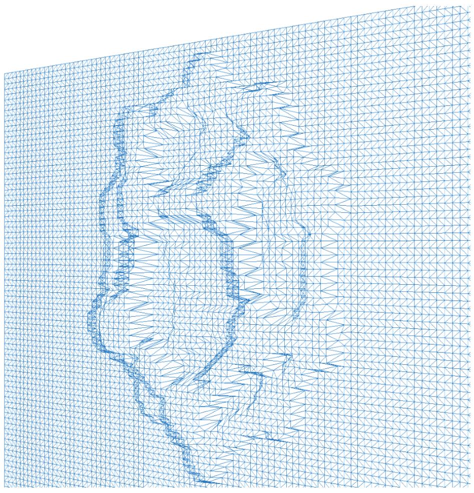

Figure 40. The z-values for each vertex of the initially planar grid are changed according to the depth map of dodecaeder.

Problems typically arise at sharp edges in the depth map. This is why this map needs to be low-pass filtered. Linear filtering can be applied natively by a GPU when downscaling a texture map, as depicted in figure 41. We choose this method because of speed, not quality.

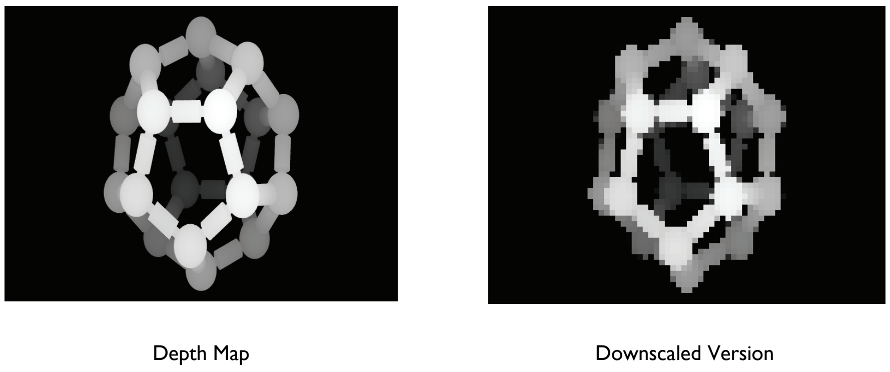

Figure 41. A depth map can be downscaled using GPU's native bilinear filtering.

We still found round-off errors. These errors become apparent when the grid is more course grain. When choosing a finer grid (e.g. 180 × 144 cells) this problem is hardly noticeable, except at sharp transition in depth. We neglect these artifacts, focusing on performance. The section 4.5 proposes a method for detecting approximations error that could possibly be extended to correct these artifacts.

4.1.4 Texture Atlases

We need at least two render targets to store all eight views and also two passes to compute its contents. We have chosen to first group four views in one render target. They represent the left hand side views. The other four views will be written to a second render target. The center view already exists as an 720 × 576 input texture. The views in each target are ordered in a 2 × 2 tiling as is shown in the deconstruction in figure 42.

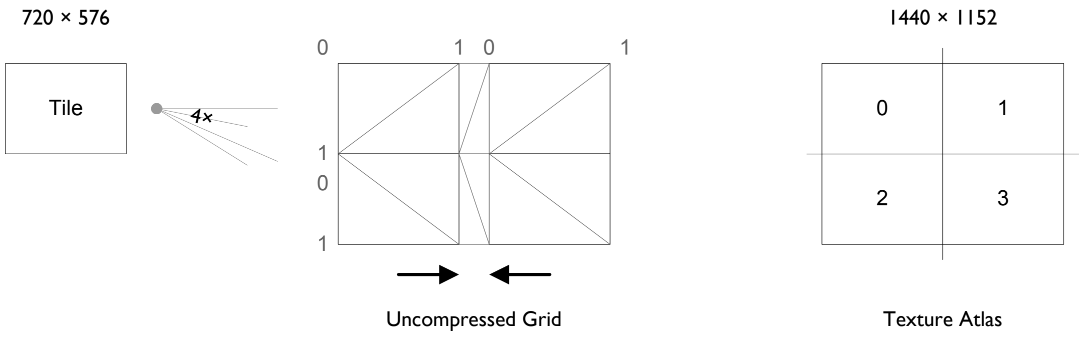

Figure 42. The texture atlas is constructed from four tiles. The in-between quads solve the problem of incompatible texture coordinates.

To prevent down sampling we chose to double the size of the two render targets. Each target has to have equal dimensions. In our case this is 2 × 720 × 576 = 1140 × 1152. The two render targets cost about 13 MB of video memory. This is within budget.

4.1.5 Merging Multiple Views

After the computation of the two atlas passes, all nine views reside in video memory; either as the input texture or in a render target. The final pass samples these inputs and constructs a lenticular output picture by interleaving (multiplexing) subpixels. This process is explained by figure 43.

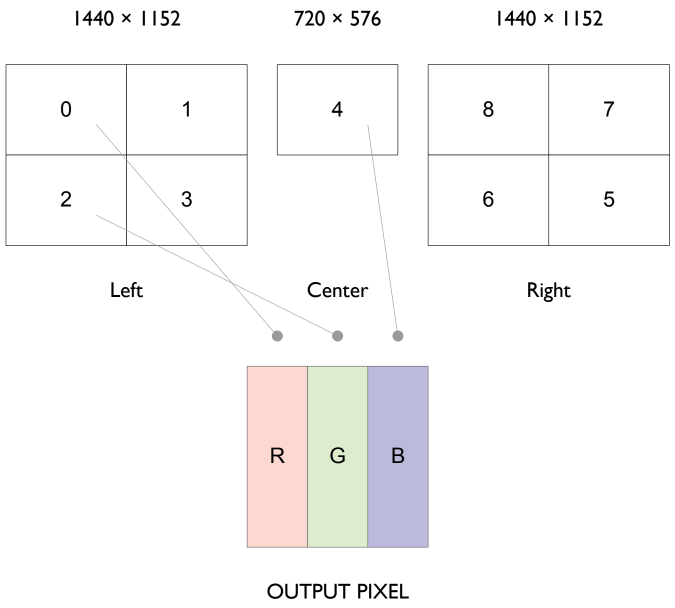

Figure 43. This example shows how the views 0, 2 and 4 are sampled when assigning the color to a certain output pixel.

No vertex manipulation is necessary in this pass so a quad suffices as geometry. Again texture coordinates define the position of the texture on the geometry surface. The fragment program first determines the corresponding view numbers for each of the three (RGB) subpixels using a texture fetch from the lenticular map. The view numbers could also be computed on-the-fly, but this is in fact just a memory versus computation tradeoff. The memory solution turns out to have a slightly better performance. The lenticular texture map holds integer values for each subpixel and these numbers correspond one-on-one with the values as define in the section 3.6.1.

The view number determines from which input texture a color needs to be fetched. In case of a non-center view, branching statements handle the texture addressing within the atlas. This shader construction depends on relatively expensive if-statements. It guarantees exactly 3 color + 1 lenticular = 4 texture lookups per fragment, so memory bandwidth usage is minimal.

4.2 Detailed Design

This section describes the design decisions for both the ray-tracing and micropolygon based rendering algorithms. Our micropolygon based shader needs a specific geometry construction in order to work. The ray-tracing solution does not depend on heavy vertex processing. We also discuss the use of textures and its filtering in this section.

4.2.1 Geometry Construction

The quality of image manipulation based on perturbing the vertices largely depends on the number of vertices defining the texture mapped surface. A more fine grain grid could describe a depth map more precisely. We want to process as many triangles as possible for our micropolygon solution.

Because of this wish, it is important to choose an efficient construction of the geometry, which allows for quick drawing. The cheapest method for drawing a 3D primitive is providing a single triangle strip. Two common drawing methods are explained in figure 44 and 45.

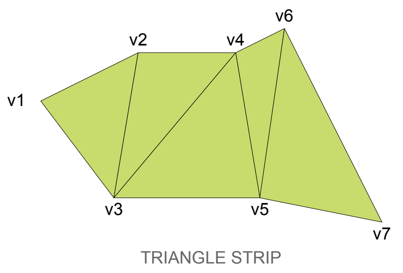

Figure 44. Vertices v1 though v7 define a triangle strip.

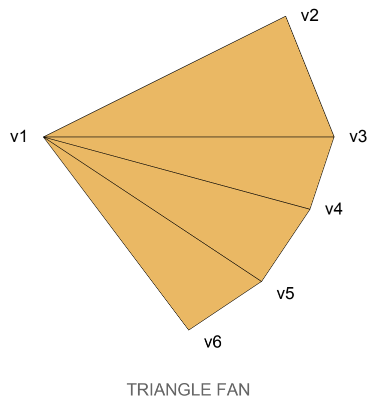

Figure 45. Vertices v1 though v6 define a triangle fan.

Specifying all three coordinates for each triangle would introduce a lot of overhead. When constructing a closed shape or object, adjacent triangles have identical coordinates. Using triangle strips or triangle fans exploit this property. Instead of fully specifying three vertices for each subsequent triangle only one vertex is defined. The process of drawing a triangle strip connects this fresh vertex with the last two vertices in the list. Drawing a triangle fan connects each fresh vertex with the last vertex and the first one in the list.

The vertices are stored in a vertex buffer, which is explicitly uploaded to the graphics board in accelerated scenarios. This also enables the re-use of geometry data without burdening the graphics bus.

As figure 46 shows, our triangle grid can be build from a single strip. Each grid cell will consist of two triangles. We must either choose to traverse the rectangular space row-first or column-first. There is no particular motivation on which approach is better. We have chosen a row-first approach; similar to reading the lines of an article.

To construct a $M \times N$ grid, where M is the number of columns and N the number of rows, we specify N rows[*]. The rows are traversed in a snake-like fashion as the example shows. Odd numbered rows a build left to right, even numbered rows right to left. This will 'make all ends meet'.

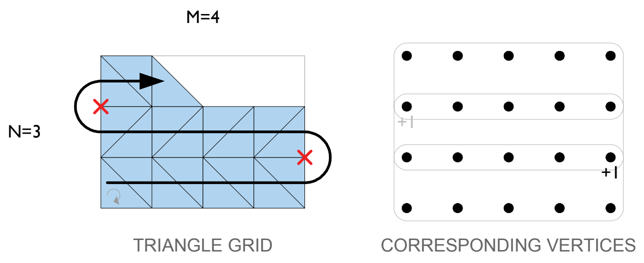

Figure 46. A triangle grid is constructed row after row.

To fill a rectangular area with $M \times N$ cells we opt to use twice that many triangles. Graphics APIs only draw triangles when their normal faces towards the camera by default. So-called backface culling can be disabled or the vertices of each triangle need to be specified in clock-wise order. We chose the latter solution, which is commonly accepted among 3D graphics specialists.

Without countermeasures every other row of the grid would get culled, because the triangle normals point away from the camera. This problem is usually fixed by adding a degenerated triangle at the end of each row. This location is marked with a small red cross in figure shown above. Such a triangle is easily created by repeating this vertex. Note that even with a triangle strip all vertices appear in the buffer twice, except for those on the first or last row.

To describe an $M \times N$ grid we have $(M + 1) \times (N + 1)$ unique coordinates. A triangle strip-compatible list of vertices is longer. Each of the N rows requires $(M + 1) \times 2$ vertices and also one vertex for the degenerate triangle, except for the last row. So, the vertex buffer needs to hold $((M + 1) \times 2 + 1) \times N - 1$ vertices. A quad results for $M = N = 1$.

Our geometry for the texture atlas adds another column to the grid right in the middle. Such an insertion only works for an even number of columns. Our texture atlases will be of at least 2×2 cells in order to support all four texture tiles. The size of the vertex buffer can be calculated by substituting $M + 1$ for M. This inserts a column in the grid. The width of this column will be zero, but it allows us to correct the horizontal texture coordinates. Row insertion is not necessary because the long sides of the rows are not connected – although adjacent – and no undesired interpolation of this vertex data will occur. The atlas buffer will hold $((M + 2) \times 2 + 1) \times N - 1$ vertices.

4.2.2 Texture Requirements

Near the start of this project it was assumed that video sources would contain 24-bits RGB data and 8-bits depth information. This motivates why the render engine is designed around these two data formats. Later it became clear, that the actual source files would be using YUV as color space with 4:2:2 subsampling. The technically correct name for the color space is YC_bC_r, which stems from PAL TV sets. The 4:2:2 subsampling means that chrominance data is horizontally multiplexed. From now on this document will treat YC_bC_r and YUV as being the same.

The YUV format encodes color using luma (Y) and chrominance (both U and V). YUV data can be converted to RGB and this can be implemented on a GPU as well. It was not our primary concern so it was decided to use CPU code. There exist multiple formulas for doing this conversion, some of which are not exact, but this has historical reasons. The conversion formula that was used can be found in appendix F.

[*] An alternative grid construction called an N-patch is not yet fully supported in the graphics driver.

Once the color data is converted to RGB it can be stored in textures, which reside either in system memory, video memory or both. These textures must be of at least the same precision. The dimensions of the source textures need to be 720 × 576.

Color and depth is stored in separate textures. This has a small performance penalty, but makes it easier to visualize and debug the rendering process. The depth map could be stored in the alpha channel of a single 32-bits RGBA texture in the future.

4.2.3 Filtering Goals

Our implementation of a custom bicubic image filter in the pixel shader allows control over the previously mentioned filter 'constant' (*a*). It is coded as a variable, which can be set in the range [−1, −1/2]. Algorithmically this mostly influences the shape of the second lobe of the cubic function. It can be used to fine-tune the sharpness of resulting image.

In the current render engine there is little to no room left in the pixel shader to do expensive custom filtering. The custom bilinear and bicubic filter were implemented as a standalone shader and were used to create some exemplary pictures for this document. We hope to incorporate a controllable cubic image filter in the future.

4.3 Ray-Tracing Shader

Patterson's and Smits' displacement algorithms inspired us to research ray-tracing of depth maps. Patterson showed how curved surfaces can be displaced along its normals. This complex method is expensive, or at least the parametric space is not easily mapped onto programmable GPUs. More so our base surface is flat, not curved. It reduces the problem to ray-surface intersection. A problem also tackled by Smits for 3D geometry.

We investigated tracing the depth map. Computing the point of intersection requires a lot of texture fetches. This is an inefficient approach and results in poor performance. Even the use of a so-called distance map does not solve this problem. The computation of such a distance map itself is already expensive. One GPU-compatible implementation uses a large 3D texture. Each volume element (voxel) of this texture stores the distance to the nearest point on the surface of the corresponding depth map. It would take megabytes of video memory to store the distance map of a single depth map. Donnelly and Du Toit implemented a demo which computes the distance map on the CPU [63]. The GPU is then used to render a displaced surface in real-time. The viewing direction can be controlled interactively. Sadly, the construction of the distance map takes a few second.

Christen concluded that directly ray-tracing a 3D scene is feasible for a limited set of objects, but did run into driver problems with the OpenGL implementation [64]. Also all objects should fit in video memory to get acceptable performance. Tracing a scene consisting of more than 60,000 polygons in a 20 × 20 × 20 space took over a second on a GeForce 6800 Ultra. We have only one object – our grid – with over one million polygons. Another problem is that we wish near 720 × 576 × 256 resolution, where 256 denotes the depth resolution.

Purcell offers two methods for ray tracing on a GPU [65]. One is a multi-pass algorithm being bandwidth limited; the other one is computationally limited. We observe that iterations must be kept low at all times. Because we trace a depth map for creating horizontal disparity we could potentially save computations. We constructed a model which computes the intersection point of a ray with the depth map for each output pixel. The *u*,*v*-coordinate of this hit is used for looking up the corresponding texel from the RGB-texture. The shader is shown in figure 48.

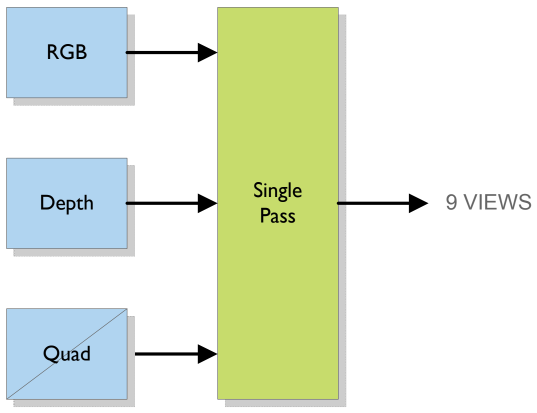

Figure 47. Ray-tracing displacement mapping uses the color (RGB) texture and depth map as well as a quad and renders nine views.

Our single pass shader needs an RGB-frame, depth map and quad on its input. The vertex program only passes on the 3D and texture coordinates of the four vertices. Almost all the work is done in the pixel shader.

The pixel shader hardware enables execution of a fragment program P, which computes

$$\langle \forall_{x,y} : 0 \le x < \mathrm{W} \wedge 0 \le y < \mathrm{H} : \mathrm{P}.x.y \rangle$$

in parallel. The constants W and H define the width respectively height of the output frame. Program P could for example do a texture fetch based on the coordinates *x* and *y*. The abstract program would read:

$$\mathrm{P}.x.y \equiv tex2D(\mathrm{colorSampler}, x, y)$$

in pseudo-code. *tex2D* represents a shader function that fetches a value from a sampler at position (*x*,*y*); in this case from the sampler named colorSampler.

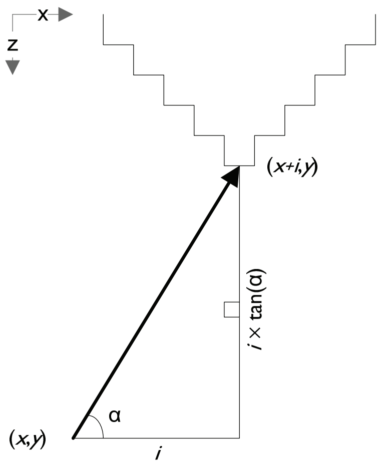

Figure 48. Ray-tracing the depth map for the output pixel at (*x*,*y*) computes the intersection point of the arrow and the surface described by this depth map. The direction is defined by angle α. The algorithm requires *i* iterations.

Our ray-tracer must also assign a single color value for each pixel. We create disparity by offsetting the *x*-coordinate depending on depth. To determine this offset, or displacement, we need to trace the depth map in horizontal direction. Figure 49 shows how to trace a ray, shot from a pixel located at (*x*,*y*) and an angle of α, towards the depth map. It shows the slice of the depth map from above. For each output we cast a ray along the appropriate viewing direction towards the depth map. As part of the iteration we make small steps along this ray (arrow) and determine whether we already passed the surface of the depth map. If so, we have a hit and can determine the *x*-coordinate. The *y*-coordinate is already known because of the scanline we are processing. We use this 2D texture coordinate to look-up the color from the RGB-texture and assign it to the output.

The ray being traced hits the surface of the depth map after *i* steps sideways. These steps are exactly the size of a pixel, because we do not want to undersample or over sample the depth map. So, we construct a for-loop with index *i* using the predicate

$$i \times \tan(\alpha) \leq \mathrm{Z} - \mathrm{D}.(x+i).y$$

to decide whether the ray has hit the surface of the depth map. Constant Z defines the upper bound along the z-axis (i.e. 256). The value of *x* + *i* turns out to be a good candidate for a displaced fetch from the color map.

Ray tracing a depth map is not any different from tracing a slice in a height field. Here the slices are scan lines. In fact we are finding the intersection point of a line with a 2D height function, which is sampled at equidistant positions; 720 at most.

The number of steps to end the recursion depends on several factors. One is the horizontal resolution, but also the resolution along the z-axis and the viewing angle matter. The horizontal resolution is 720 pixels and an 8-bit value is used to describe the pixel's depth. The maximum simulated viewing angle was not specified for this project. It was advised to try to support up to 30 pixels of horizontal disparity, but 30 fetches already exceeds the bandwidth budget.

The problem is the number of texture fetches. Each iteration step of the tracing algorithm needs to read a value from the depth map. This limits our shader performance to the video memory bandwidth. High-end programmable GPUs have up to 35 GB/s of total bandwidth, which is roughly 1 GB per rendered frame. It does not mean we can fetch 1 megapixel textures over one thousand times. Research showed we could fetch about 10 texels per output pixel, where we would expect higher numbers considering the total video memory bandwidth. Perhaps this is due a disadvantageous texture cache strategy, or ill performing scheduling of fragment processing.

4.4 Micropolygon Based Shader

The micropolygon algorithm for displacement mapping turns out to be the most in line with the architecture of programmable GPUs. The algorithm relies on a heavily tessellated geometry. New hardware can handle lots of triangles. Small features of the geometry can then be manipulated. This task is suitable for the vertex shader. Thereafter the color texture is applied, which can be done by the pixel shader very fast. Figure 47 shows the design of our micropolygon based shader.

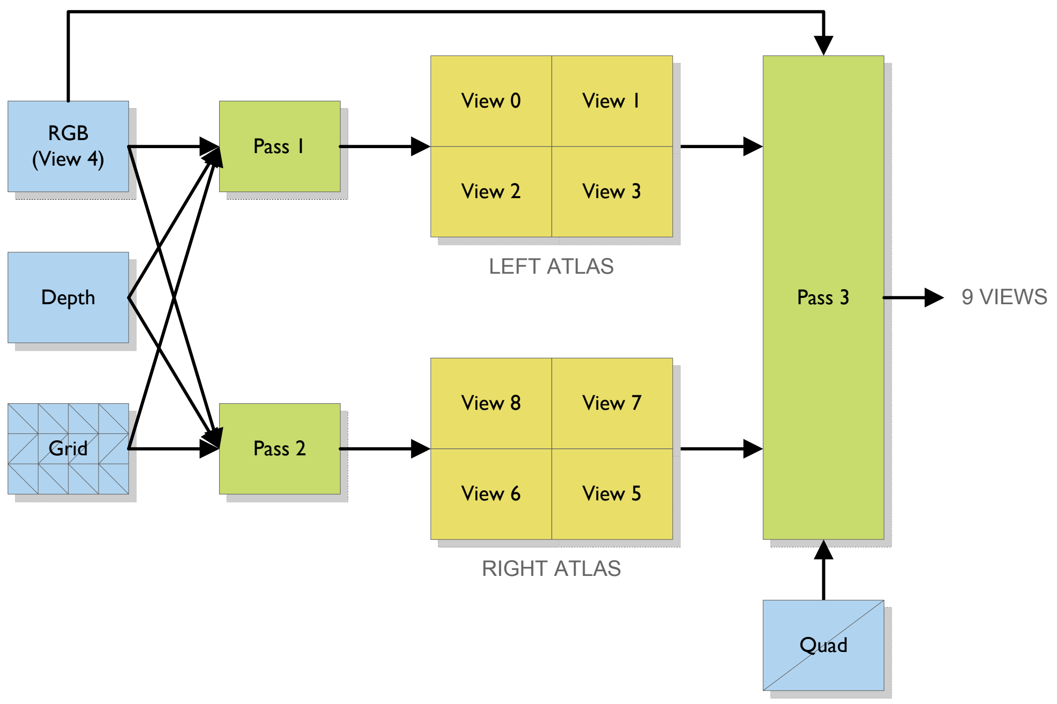

Figure 49. Micropolygon based displacement mapping uses three passes to compute and merge nine views.

This shader uses three render passes to generate nine views. Each pass always implements both a vertex program and a fragment program. The former code is executed on the vertex shader; the latter on the pixel shader.

Pass 1 needs the RGB-frame, corresponding depth map and unaltered geometry. The pass creates four views for the left-hand side and writes the output to a render target. Pass 2 does the same for the right-handed views. This output is stored in the second render target. Both passes depend on a fine-grain grid of typically 180 × 144 cells[*]. The hardware should be capable of processing

$$(2 \text{ triangle/cell} \times 180 \times 144 \text{ cells} \times \text{atlases/frame} + 2 \text{ triangles/quad}) \times 25 \text{ fps} \approx 2.6\text{M triangles/s}.$$

Pass 3 combines the original RGB-frame and the eight pre-processed views from the render targets in a lenticular formatted image. A quad consisting of two triangles suffices for this mapping.

Although GPUs allow rendering to multiple targets from a single pass, we will not. Pass 1 and 2 require a different viewing direction – one from the left and one from the right – and thus the projection matrices differ. Matrices cannot be changed within a render pass.

4.5 Improved Displacement Shader

The following design allows detection of the approximation error, resulting from the grid fitting of the depth map. Low-pass filtering the depth map removed aliasing problems and provided better results. We even found that the under sampling of the depth map is not that big of a problem in case of real-life footage. The estimated depth from such sequences tends to have less sharp variations in depth.

We succeeded in writing a real-time shader that superimposes a distinct color on the areas with a large approximation error. This is a good starting point for reducing the remaining artifacts from our micropolygon based shader. We would like to attempt ray-tracing the limited areas that have a large error.

[*] The results of different grid configurations can be found in section 6.1.

The schema in figure 50 shows the structure of the multi-pass algorithm which can detect grid artifacts. Pass 3 and 6 could implement some kind of texture correction on the problem areas.

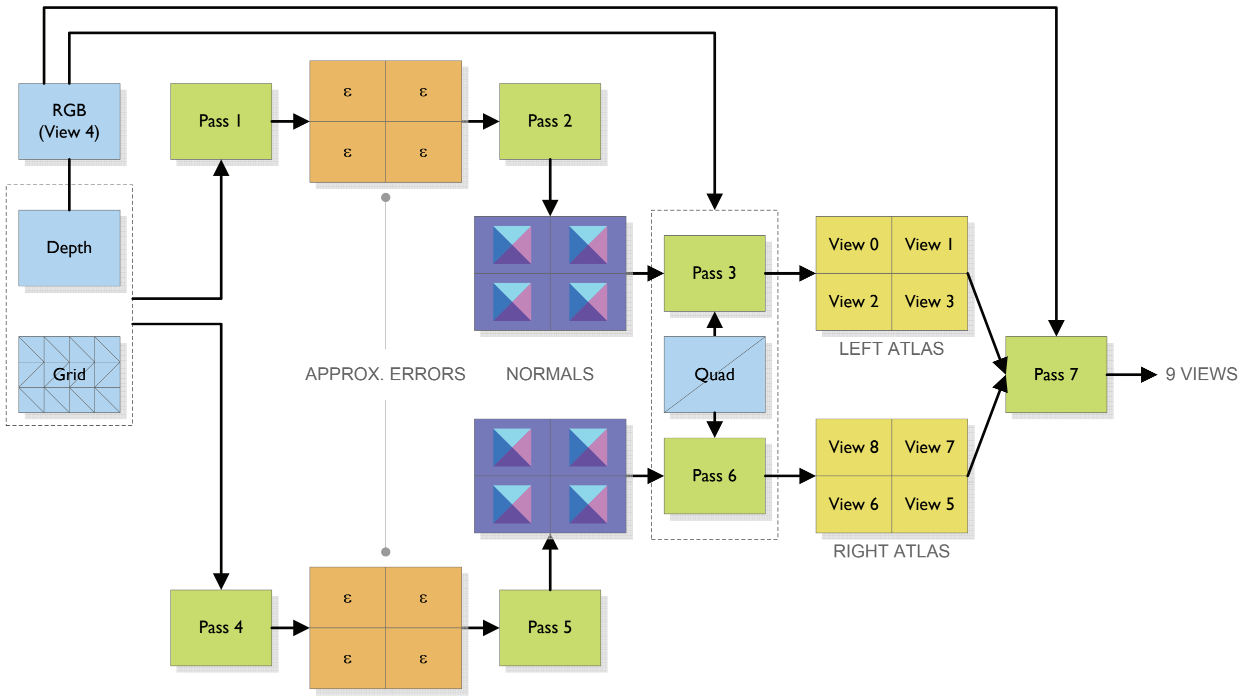

Figure 50. Texture corrected displacement mapping requires additional passes (1 & 4) to compute the size of the approximation error (ε) for each output. Passes 2 and 5 compute normal maps that could help the final correction passes.

Pass 1 and 4 compute the error between the per-pixel depth approximated by the triangle grid and the actual depth, as specified by the depth map. We let these errors determine the amount of correction that must be applied in one of the following passes. The sign of the error is also required. This is why we construct per-pixel normal vectors, which are computed by pass 2 and 5. The size and direction of these vectors can be used to visualize and potentially correct artifacts, which are the result of imperfect grid fitting. However, a solution based on this schema has not yet been found. The texture corrections applied in one place still introduce artifacts in other places. We did succeed in visualizing the fitting errors of displacement mapping.

5 Implementation

The development of a render engine for driving a display device involves a selection of software tools. We would need the usual development equipment like an editor, compiler and debugging aids. Because we were to design a graphics processing application for a programmable GPU we also needed to decide on appropriate graphics and shader tools. This chapter motivates our choices.

5.1 Software Selection

The primary driver for the software selection was the exploitation of new programmable graphics hardware for PCs. We looked for hardware and software combinations that would support new technologies for rendering during the first months of the project. Not all programmable GPUs were suitable, mostly because they lacked some shading features.

New hardware features can only be used when they are exposed by some software interface. Nowadays the operating system prevents almost all direct access to hardware for security and stability reasons. Software interfaces offer a set of functions which are implemented in the corresponding software layer. The layered structure is what programmers call a stack. Figure 51 is an example of a stack for graphics processing [66].

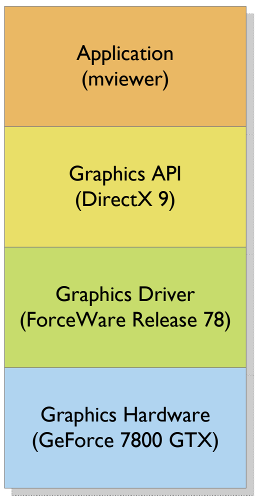

Figure 51. The graphics stack shows the relation of the demo application, graphics API, device driver and video hardware.

At the bottom we have a hardware device, such as a sound card, wireless adapter or in our case a video card. On top of that sits the device driver. It is part of the machines operating system (OS) and enables the use of this piece of equipment. This software level separates hardware specifics from the rest of the OS. Another level of the software stack implements generic functionality which is available to the application developer, for example DirectX or OpenGL. Application software often builds onto a multitude of APIs. Our concern is a suitable graphics stack. The application, which is on the top level, interfaces with the end-user. In a windowed environment it exposes a graphical user interface (GUI) and allows interaction.

We investigated programmable features of GPUs from three mayor manufacturers, being ATI, NVIDIA and Matrox. We used the graphics drivers from NVIDIA (ForceWare release 78.01), which should work identical on third party boards. The drivers are more current and offer then most functionality, especially related to shader programming.

Back in June 2005 only NVIDIA's 7800 model and ATI's X600 and higher models supported the newest shader model (3.0) in both DirectX and OpenGL. This shader model requires the use of either OpenGL 2.0 or DirectX 9. OpenGL exposes the shader hardware features via the extensions `ARB_vertex_program` and `ARB_fragment_program`[*]. There also exists non-approved (often vendor specific) extensions, but these would introduce uncertainties to the project.

DirectX offered the features of the new shader model in one clear interface and offered them a bit earlier than OpenGL did. Also, a lot of programming examples were based on Microsoft's API. The logical choice was to go for a DirectX 9 capable card. We selected a GeForce 7800 board, so we could also revert to an OpenGL-based implementation.

It turned out that NVIDIA's tools were very useful for shader development. We decided to construct our vertex and fragment programs in FX Composer and would develop a C++ DirectX application in Visual Studio, which could demonstrate these shaders. FX Composer enables fast prototyping of shaders. Visual Studio supports shader debugging since its 2002 .NET release.

5.2 Shader Models

Today, already three generations of shader models exist. The fourth generation will be incorporated in DirectX 10, which will be part of Microsoft's next operating system, Windows Vista. Each shader model (SM) defines feature requirements and specifies a set of instructions for graphics hardware. It defines what the vertex- and fragment processors on the video card must be capable of. The requirement details on register counts and other features can be found in appendix D. This specification is the merit of Microsoft and can be consulted at Microsoft's Developers Network[†] (MSDN). Next, we summarize how the shader models evolved.

SM 1.0 features a basic instruction set to manipulate register values. Loading, adding, subtracting, copying and comparison operations are available. The set also enables logarithmic calculus – in both full (32-bit) and partial (16-bit) precision – and allows typical graphics arithmetic like dot product, matrix multiplication and lighting. All instructions have an associated cost expressed in a number of slots. The `min`, `max` and `mad` (multiply-and-add) instructions all use one slot.

The 1.X pixel shader instruction set enables the same basic operational codes (opcodes) for arithmetic. These are `add`, `dp3`, `dp4`, `lrp`, `mad`, `mov`, `mul` and `sub`. The initial model for the fragment processor included access to texture memory. This is reflected in various texture instructions. The most important texture fragment operations are loading (`texld`) and killing (`texkill`), the latter which causes the pixel to be excluded from rendering.

SM 2.0 adds boolean and integer registers and accompanying instructions `defb` and `defi`. Basic flow control instructions were included in the (vertex shader) set. The assembly codes for selection blocks are `if bool`, `else` and `end if`. Loop constructions have `loop` followed by `end loop` or use the repetition codes `rep` and `endrep`. The cross product can by calculated using `crs` and two values can be linearly interpolated by `lrp`. Both operations cost two VS slots. `sng` determines the sign of register value and a vector is normalized by `nrm`; both operations at the cost of three slots. x to the power of y – where x and y are two registers – is calculated by `pow` and the sine and cosine are retrieved by `sincos`. The result of the latter instruction is returned in radians and costs eight slots. `abs`, `call`, `callnz`, `ret` and `mova` are also new to SM 2.0.

Pixel shader version 2.0 adds the same matrix multiplication as supported in vertex programs to the pixel programs. This also holds for the `abs`, `exp`, `log`, `max`, `min`, `nrm`, `pow`, `rcp`, `rsq` and `sincos` instructions. New fragment shader specific operations are `frc`, which returns component fractions,

[*] ARB stands for Architecture Review Board; the consortium that approves OpenGL extensions.
[†] The Microsoft Developers Network web site is located at http://msdn.microsoft.com/

and two texture operations, `texldb` and `texldp`. The last two codes are used for biased respectively projected loading from a texture. `dcl` was added to declare the newly available pixel shader registers, optionally in partial precision (`_pp`). `dcl_samplerType` is used to declare 2D, cube or volume samplers (denoted by `s#`).

SM 2.X extended the vertex shader instruction set with `break` and a few predicate operations. These are `setp` to set the predicate register, `breakp` and new versions of the conditional codes `callnz` and `if`. PS 2.X added breaking, call-return and conditional codes to the instruction set. Conditions, boolean expression and predicates can be used to select the appropriate execution path. The new boolean and integer constant registers are defined by `defb` and `defi`, analogous to VS programming. Version 2.X also added repetition and marker labels. Up to 16 labels were supported, but can only be used directly after a `ret` code. `dsx` and `dsy` extract the change in x- and y-direction (gradient) of the render target.

SM 3.0 added the instruction `texldl` to the vertex shader set. It is used to load a texture with user-adjustable level of detail (LOD). An important improvement is that the texture fetch instructions could also be used in vertex programs. This feature is called vertex texture fetching. As of SM 3.0 the vertex shaders are no longer limited to geometry and constants for its input. Mathematical functions can also use texture data for geometry manipulation in a way similar to what fragment code can do for setting a pixel's color. Finally, the 3.0 pixel shader adds the `loop...endloop` construction and also features the `texldl` LOD texture lookup instruction.

We need shader model 3.0 to implement our render engine, because the engine requires vertex texture fetches (VTFs). Also compiling fragment program for view multiplexing results in over 64 arithmetic shader instructions, which requires at least the same shader model on NVIDIA cards.

5.3 Shader Languages

In modern graphics programming, one hardly ever needs to code in assembly, as discussed in the previous section. The exception is general purpose programming of the GPU; often referred to as GPGPU.

The use of so-called high level shader languages eases the task of coding and reduces the number of errors. Today's shader program compilers are actually quite good, in the sense that assembly coding by hand shortens the programs only by a handful of instructions. This could well be caused by the fact that the size of the instruction set is still well under 80 operation codes. This optimization could be important, however the time spent on developing the shader in assembly is much higher.

Various shading languages exist today. These are Microsoft's HLSL, NVIDIA's Cg and for example Sh, which extends C++. Another GPU programming languages are the Brook stream language, which is developed at the Stanford University [67], and GLSL, which is managed by the Architecture Review Board. We will shortly discuss three high level shading languages. These high level languages all offer useful functions for geometry transformation and lighting. This includes normalized 2, 3 and 4-component vector functions such as reflection, refraction, clamping and clipping. Other features are 4×4 matrix multiplication, trigonometry and texture lookup.

5.3.1 Cg

Cg stands for C for Graphics [68]. The specification for this high-level shading language was developed by NVIDIA. Cg is an open standard for shader programming. NVIDIA encourages others to write their own compilers for this language, but none have decided to do so until this date.

Cg is designed to work with both DirectX and OpenGL. DirectX 8 is support, but the primary focus is on version 9. The Cg compiler produces shader assembly code for various GPU architectures and platforms, compatible with both OpenGL and Direct3D applications. The target platform profiles are selected the compiler switches of `cgc.exe`. This compiler is included in the Cg Toolkit, which is available for the Windows, Linux and Mac OS X platform.

NVIDIA also provides a set of development tools for content creation (i.e. FX Composer) and profiling (i.e. NVPerfHUD). FX Composer is suitable for shader development and prototyping. It uses Microsoft's effect compiler and not the before mentioned shader compiler.

5.3.2 GLSL

The shading language for OpenGL has a more restricted set of data types; half~ and double precision types are absent in this specification [69]. Matrices in GLSL must be of equal dimensions. Effect files for this shading languages are not compiled using a standalone tool. The actual conversion to GPU assembly is supposed to be built into the graphics hardware drivers and thus the responsibility of the independent hardware vendors. This welcomes innovation through flexibility, but could also result in stability issues. This design decision also means that hardware profiles – like HLSL and Cg have – do not exist.

Developers are expected to select appropriate OpenGL extensions themselves. These extensions come in both vendor specific and non-vendor specific (ARB) flavors. The latter are recognized by the extension's name `_ARB_` infix. GPU programmers can start using shaders by including the object extensions `GL_ARB_shader_objects`, `GL_ARB_vertex_shader` and `GL_ARB_fragment_shader`. The common programmable features reside in the accompanying `GL_ARB_vertex_program` and `GL_ARB_fragment_program` extensions. Both shader and program pairs can be compiled to GPU assembly by the graphics driver. Additional functionality is offered through vendor specific extensions such as `ATI_fragment_shader` and `NV_fragment_program`[*].

Features implemented by these extensions became available somewhat later than its DirectX counterparts. Only the newest card of NVIDIA supported OpenGL version 2.0 back then, as well as a few cards from ATI. This is also the reason for choosing DirectX. Our current implementation could be ported to OpenGL. The shading concepts remain the same.

5.3.3 HLSL

By many regarded as the most mature and stable solution for GPU programming, HLSL is tightly integrated into DirectX and thus limited to the Microsoft's Windows platform. HLSL code closely resembles ANSI C code but adds specific functions and compound data types, just like Cg and GLSL do. HLSL is not an open standard.

Shader programs are compiled offline using `fxc.exe` or at runtime. Microsoft also provides an effect editor as part of their DirectX 9 SDK. This editor only handles one shader at a time and does not provide preview windows for textures and render targets. It also lacks color coding, which makes it less attractive for shader development. We conclude this section with table 10, which provides an overview of three high-level shading languages.

[*] The OpenGL Extension Registry is published at http://oss.sgi.com/projects/ogl-sample/registry/.

Shader Language	Cg	GLSL	HLSL
Vendor	NVIDIA	ARB	Microsoft
Supported Platforms	Linux, OS X, Windows	Linux, OS X, Windows	Windows
Compilation & Linking	DirectX, OpenGL	Hardware driver	DirectX
Scalars	bool, int, half, float, double	bool, int, float	bool, int, half, float, double
Vectors	2-4	2-4	2-4
Matrices	M × N	N × N	M × N
Samplers	1D, 2D, 3D, cube map	1D, 2D, 3D, cube map, shadow map	1D, 2D, 3D, cube map
Combine User-defined Functions	No	?	Yes
Intellectual Property Protection	No, must compile into exe	No, must compile into exe	Partially, through assembly shader stream format

Table 10. This table summarizes the properties of three high-level shading languages; deduced from Lovesey's technical report [70].

5.4 General Framework

Figure 52 shows the global structure of our demonstrator application. The main class is located at the top. All customary classes are depicted in yellow. The interfaces are shown in blue and its components in green. The arrows represent (the direction of) function calls.

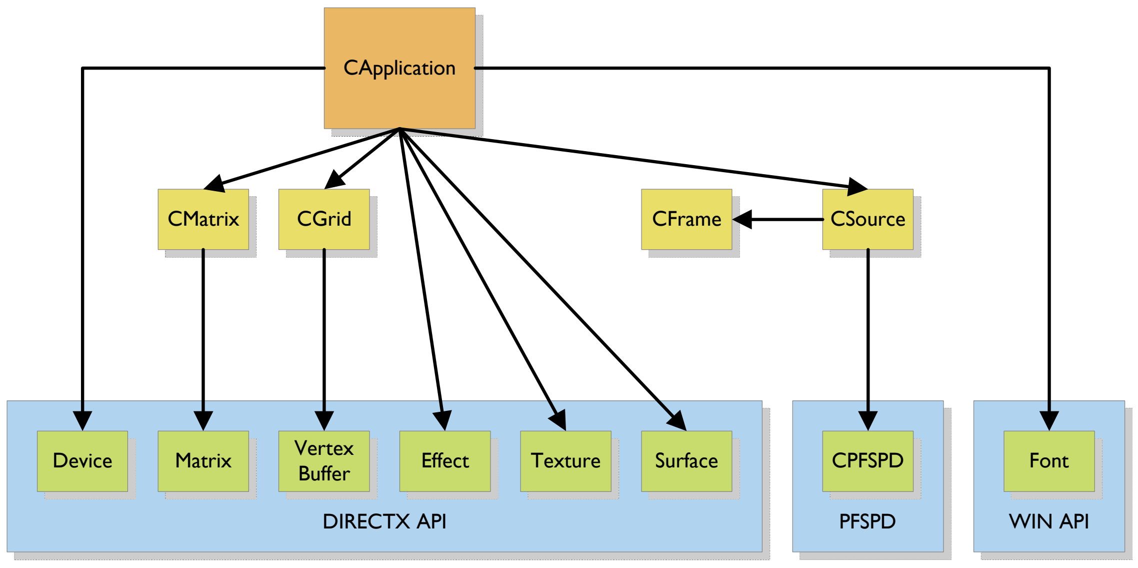

Figure 52. This application overview depicts both the dependencies of the internal classes and its external interfaces.

A top-level class (not depicted) implements the window and menu creation, message loop and instances a single `CApplication` object. Window messages result in function calls to this application object.

The `CApplication` instance not only holds the main state variables but also controls the primary DirectX device and the additional graphics resource for effects, textures, surfaces and a font object for overlay text. The application class also manages the instances of the matrix, grid and source objects. These customary classes use the interfaces of DirectX matrices, vertex buffers and the CPFSPD library. The font interface is part of the common Windows API. `CApplication` also implements the render loop (`Render()`) and the frame preparation (`PrepareFrame()`), which updates the appropriate textures in video memory.

During runtime three instances of `CMatrix` live. They store the current values for the world, view and projection transform.

The application also uses three vertex buffers. The buffers define the geometry for a texture quad, a textured M × N grid and texture atlas. The structures are programmatically created by the `CGrid` class, not loaded from a file.

A single `CSource` object is responsible for loading video data from disk. The actual disk operations are implemented in the CPFSPD library, which was written in ANSI C. `CSource` serves as a wrapper between the PFSPD interface and the main program. Both color and depth instances of `CFrame` are used for intermediate video frame data; one frame to store Y, one for U/V and another one for D. The color data is provided in YUV format and needs to be converted to RGB before it is used by our render engine. This color space conversion is implemented as CPU code. Otherwise it would have tainted our shader code. This design change could perhaps be incorporated in the future. The next sections explain our implementation in more detail.

5.4.1 Naming conventions

The name of each class is prefixed with a capital letter 'C'. The C++ source code for each of these object classes resides in a header (`.h`) and implementation file (`.c`). The filenames of effect files all have the `.fx` extension. All filenames for this project's source are in lowercase.

Variables are named conform the 'Systems Hungarian' notation. It is a coding convention for commenting source code. This includes the naming of variables by their data type and its scope. Table 11 provides a limited overview.

Data type	Prefix
character	ch
dword	dw
handle	h
int	n
pointer	p
word	w

Scope	Prefix
class member	m_
global	g_

Table 11. This table was compiled from Simonyi's article [71].

This writing style certainly has some objections when it come to finding program bugs as Spolsky pointed out [72], but can make code more readable, mostly because Microsoft's reference documentation uses this convention. We considered it to be an acceptable choice as long as the style is applied consistently. Using the so-called 'Apps Hungarian' convention would have been better.

5.4.2 Initialization

After the creation of the program window a few additional initialization steps are carried out. The Direct3D object is created. This step also checks whether the correct DirectX runtime library exists on the client machine. Next, the actual hardware capabilities of the graphics boards are determined. This step checks for the availability of hardware vertex processing. A render device is created. Render devices of debug builds support the NVPerfHUD profiling tool (on the primary display). Multi-monitor support is available in non-profiling situations; either in windowed or full screen mode.

During initialization the appropriate font size parameter is determined and the `CFont` object is created. A timer object is set for the computation of the FPS display. The custom vertex format is declared, which is described in the section named 'Input Streams'. The initialization is ended by spawning a list of resource initialization procedures. These are `InitD3D()`, `InitVB()`, `InitTextures()` and `InitEffects()`.

Resource Creation

`InitD3D()` applies suitable value for the world, view and projection matrices and sets the render states of the graphics pipe: lighting, z-buffering and culling is disabled for the fixed pipe. The shading mode is set to flat shading instead of the Gouraud default. This is appropriate because lighting is already disabled.

Our shader demo depends on three geometric models being a quad, a grid and the model for our texture atlas. `InitVB()` creates each of these three models in a vertex buffer and uploads them into video memory.

For texture resource creation the requirements are checked against the devices' capabilities. E.g. the engine depends on multiple render targets and needs support non-power-of-2 textures. The `InitTextures()` procedure is ended by setting sampler states and configuring the texture stage. The fixed function sampler is set for texture clamping and linear filtering. The texture stage will use texture mapping without any fogging or blending.

`InitEffects()` loads and compiles the corresponding effect file for the current shader mode. Each shader uses a specific set of parameters such as the amount of depth, depth offset, but also one or input textures. These parameters are linked to application variables directly after successful compilation. In case of an error this procedure shows a warning message and causes a fallback to fixed function rendering.

Render Capabilities and the Adapter Menu

The top-level menu of `mviewer.exe` always contains the items File, View, Play, Window and Help. Another item – the Adapter menu – is inserted at runtime. This menu is context sensitive. The entries in the adapter menu depend on the capabilities of the graphics card. The menu lists the video adapters and available display resolutions. Rendering to a lenticular display normally requires rendering at full resolution (i.e. 1600 × 1200 or 1920 × 1080). The `DetectRenderCaps()` procedure determines the set of features offered by the graphics hardware and modifies access to the shader modes accordingly. The adapter menu is initially built by procedure `InsertAdapterMenu()`. After the initialization steps the program enters its main loop.

5.4.3 Main Loop

The main loop contains the message loop and the render loop. The former handles the window messages. Window messages result from the user's keyboard, mouse events and more. The render loop implements all DirectX related drawing.

5.4.4 Input Streams

Apart from the uploaded vertex and fragment program, the graphics board has two types of input streams. One kind is geometry. It includes the quad, grid and atlas structures that are loaded into video memory during initialization. The other input stream is the video data; both for color and depth. These are concurrent streams. Before a GPU can interpret the data it receives these information streams need to be defined on the application side. A DirectX implementation is highlighted next. This code will instruct associate the data streams with certain memory registers, e.g. vertex data will end up in the input registers of the vertex shader.

The application header file (`application.h`) defines our vertex format in two different methods. The older method uses a C-style structure.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;//The untransformed position of the vertex
    FLOAT u, v;//The (2D) texture coordinate
};
```

The other method uses the Flexible Vertex Format (FVF) and allows customization of how this vertex data is streamed to the graphics hardware.

```
#define D3DFVF_CUSTOMVERTEX ( D3DFVF_XYZ | D3DFVF_TEX1 )
```

The vertex declaration is as follows:

```
const D3DVERTEXELEMENT9 declaration[] =
{
    {0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
    D3DDECL_END()
};
```

This declaration announces that each vertex element in stream 0 starts with 12 bytes to specify a position in 3D space and that the next 8 bytes define a 2D texture coordinate for this vertex.

The flow configuration of geometry data is now prepared, but we also need to direct two video streams. We read our YUV and depth frames in two separate calls. The current CPFSPD library does not offer a function that combines the two. This also has a somewhat negative effect on the performance when reading from disk, where seek times are relatively high. When the data is fetched the YUV frame is converted to a DirectX RGB surface using the computation described in appendix E. The depth information is copied to an L8 texture, which represents 8-bit luma. Then, both DirectX surfaces are uploaded to video memory. This operation overwrites the existing textures for color and depth.

5.4.5 Shader Loading and Interfacing

In DirectX effect files can be loaded and compiled at run-time using `D3DXCompileShaderFromFile`. We choose this method over pre-compilation, because it allows one to edit the shader text file while running the render application. Saving changes and reselecting the effect immediately shows the impact of the revision. Also, shader compilation is fast, especially compared to application compilation. We designed a strict interface between our effect files and the application. This enables the sharing of effect parameters controlled by the application. The following variable names are defined:

- `g_ColorMap`
- `g_DepthMap`
- `g_LenticularMap`
- `g_Effect`
- `g_Offset`

The first three variables provide access to the color, depth and lenticular map. `g_Effect` and `g_Offset` allow control over the amount of effect and an offset by the user. For our première multiview shader, `g_Effect` controls the amount of disparity. `g_Offset` determines the origin on the z-axis. The semantics of these variables could also be chosen differently. Some of the featured shaders do not have any controllable parameters, e.g. the texture mapping shader.

5.4.6 *Lenticular Mapping*

The next simplified code snippet shows the implementation for filling the lenticular texture map with the appropriate view numbers.

```
D3DLOCKED_RECT rect;
m_pLenticular->LockRect( 0, &rect, NULL, D3DLOCK_DISCARD );
UINT *pfBits = (UINT *) rect.pBits;
UINT view = 0;
int row = 0;
for( UINT i = 0; i < (m_nTargetHeight * m_nTargetWidth); i++ )
{
     if( i % m_nTargetWidth == 0 && i != 0 )
     {
          row++;
          view = 0;
     }
     //RGB-bits
     UINT R = ((view * 2) + 9000 - row) % 9;
     view++;
     UINT G = ((view * 2) + 9000 - row) % 9;
     view++;
     UINT B = ((view * 2) + 9000 - row) % 9;
     view++;
     pfBits[i] = (R << 16) + (G << 8) + B;
}
m_pLenticular->UnlockRect( 0 );
```

Before writing to the texture buffer the memory must be locked. A for-loop sequentially fills the buffer pixel after pixel and row after row. A single index variable (`i`) suffices because consecutive rows are mapped adjacently in memory. The constant 9000 represents a big enough variable belonging to the class of X modulo 9 = 0. This number must be larger than the maximum height of the lenticular texture, which is usually 1080 or 1200 pixels. Otherwise the intermediate value could become negative, which is illegal for unsigned integers (`UINT`). Two bit-shift-left operators (<<) put the values for the red and green subpixel into place. Finally, the texture is unlocked again.

5.4.7 *Render Loop*

Each execution of the render loop computes a full output frame. It is always wise to keep this routine streamlined. We moved all the possible state changes out of the render loop the keep overhead to a minimum. 'Aggressive' debugging (warning level 4) still shows a few unnecessary state changes. Further optimization is possible using the DirectX EffectStateManager interface. It allows custom control over the state changes of the graphics hardware such as render states (`SetRenderState`) and texture stages (`SetTextureStageState`). Performance gains will be low as the number of state changes we monitored is less than 20. Currently, state switching is not a bottleneck.

6 Results

We succeeded in designing and implementing a demonstrator for the nine-view display device on a GPU. The demonstrator provides a 3D impression from RGBD video. This chapter presents the results for the micropolygon displacement shader, which is the image processing engine of the demonstrator. We performed three series of tests to determine the performance bottlenecks of this shader. The three series were repeated on three different video cards, being NVIDIA's 6200, the 6600 GT and the 7800 GTX. Running tests on different hardware allowed us to get a better understanding of how shader hardware specifications translate to frame rates and also revealed adequate program settings.

Potential bottlenecks for render engines include the communication bandwidth to (and from) the graphics card and the speed of its video memory. The vertex or fragment shader hardware could also limit performance. The bandwidth provided by PCIe and AGP is sufficient to sustain the upload of the 720 × 576 RGBD stream.

The measurements for the 7800 GTX graphics processor were run on an Intel Pentium 4 3.0 GHz system with 1.0 GB of memory running Windows XP. The 6600 GT and 6200 tests were run on different machines, but of comparable specification. The frame rates were computed by the demonstrator itself, but also confirmed by using NVPerfHUD, a GPU profiling tool from NVIDIA. The following paragraphs discuss the results in more detail. The performance measurements are tabularized in appendix E.

6.1 Vertex Processing

Our algorithm requires a huge amount of vertex processing. The computation of displaced vertices is the bottleneck of this system when rendering at the full output resolution of 1920 × 1080 pixels. Ideally we could manipulate a textured grid consisting of 1440 × 1152 vertices in order to speak of true micropolygons. Such polygons are smaller than a single texel.

The algorithm can push the vertex processing to about 47M displaced triangles per second on a 7800 GTX and 18M on a 6600 GT, when using 2 × 2 textures. NVIDIA states that the 6800 can process up to 33M displaced vertices per second [73], which is about the same in triangles when they are drawn in a single strip. This makes sense because the 6800 has only 6 compared to the 7800's 8 vertex processors. Realistic texture sizes result in a performance of 33M displaced triangles per second.

We tested various grid configurations ranging from 90 × 72 cells up to 720 × 576 cells. This changes the load on the limiting vertex shader hardware and thus affects performance. Dimensions below 90 × 72 did not cause a bottleneck. At 720 × 576 the throughput always dropped below 25 frames per second, even using the 7800 GTX, so we did not measure beyond these dimensions. Figure 53 shows the frame rates we measured.

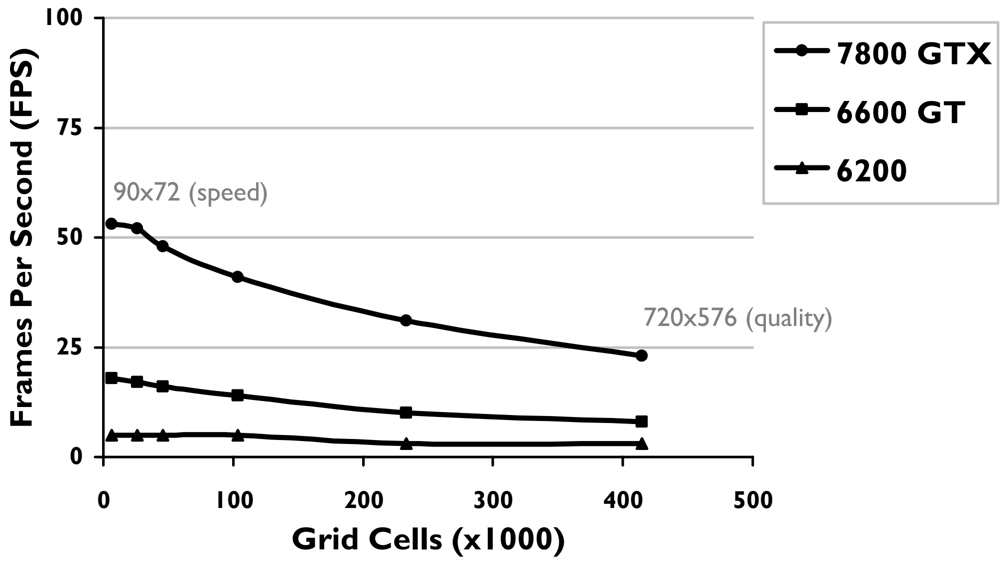

Figure 53. The frame rate (FPS) gradually drops when rendering 1920 × 1080 on a 7800 GTX while increasing the number of grid cells.

The number of grid cells affects the final image quality. Although the images remain sharp, lowering the grid dimensions worsens the depth perception. The reason is that the geometry can not represent the depth map that well, when a lower number of triangles is used. On the 7800 GTX, we were able to sustain a frame rate of just over 25 frames per second when using a grid of 360 thousand cells. This holds for both an equally dimensioned 600 × 600 grid as a 692 × 520 grid, which has square cells when the aspect ratio is 4:3. Such grids result in acceptable stereoscopic quality.

We need more than eight vertex pipes, a 7800 GTX offers, to improve the picture quality any further. It is uncertain whether video cards for the professional market perform any better than consumer cards when it comes to vertex processing. Professional cards have the same number of vertex pipes, although they offer some additional hardware accelerated features for computer-aided design and digital content creation. However multiview rendering is unlikely to profit from these features. More so, we do not expect that upscaling the grid dimensions will become feasible any time soon. Single GPU workstations do not have the power to process, for example, four times the number of vertices, when doubling the grid dimensions to 1440 × 1152. We will have to look for other means to improve on picture quality. Feature correction implemented on the pixel shader, as proposed in section 4.5, could be a solution.

6.2 Pixel Shading Performance

To get the best results we kept the number of texture fetches at a minimum. Nonetheless the lenticular compatible output requires different views for each subpixel. Our render engine is capable of rendering at any output resolution the graphics board and display device advertise via DirectX.

Higher output resolutions will result in lower frame rates due to video memory bandwidth limitations. Lenticular displays only have one native setting where the 3D picture is correctly aligned with the lenses. Different settings will not result in pictures that give a 3D impression. This means we cannot save on the output setting and we are obligated to render at 1920 × 1080 pixels. In figure 54, we provide the test results when using a 3 × 2 grid. It helps us predict whether the render engine will be sufficient to drive multiview devices with different screen dimensions.

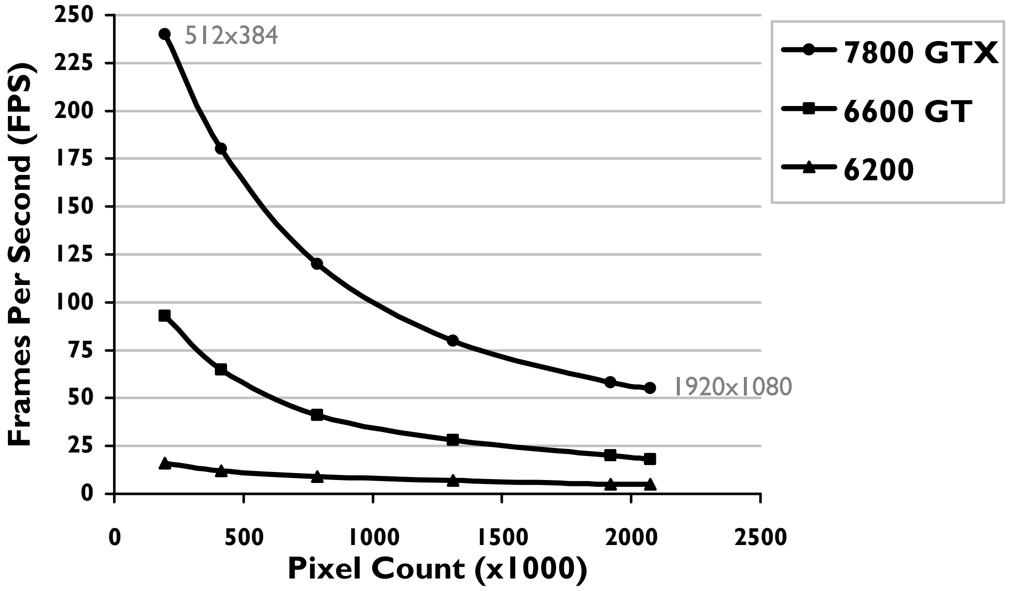

Figure 54. The frame rate (FPS) gradually drops when increasing the output resolution. To minimize the load on the vertex shader a grid of 3 × 2 cells was used.

On a 7800 GTX, we are able to multiplex nine standard definition views at just over 50 frames per second at the maximum output resolution of 1920 × 1080 pixels. When we use finer triangle grids, this number drops to just below 25 frames per second. The cost of fragment processing increases progressively with the total number of triangles that are being rasterized to create these fragments. Thus, more triangles cause more overhead.

The technical development of memory chip, regarding speed, tends to increase slower than the technical development of processors on the market for PCs. We should not expect (3D) output images getting bigger than 1920 × 1080 in the near future. We think the extra power of newer computer hardware is better spent on improving depth estimation and advise to stick to rendering at current high definition output format.

6.3 Image Filtering

It is hard to quantify the results for image filtering. We can state, however, that we successfully implemented custom filtering on programmable GPUs, be it as a separate shader. We coded a bilinear and bicubic image filter. The sharpness of the bicubic filter can be controlled by a parameter. Comparative pictures can be found in section 3.7. For optimal speed of custom filters built-in linearly interpolated texture look-ups could be exploited [74]. Still, custom filtering is not used in combination with our micropolygon displacement shader because it has a negative impact on the frame rate. Currently, this combination of multiview rendering and filtering is too expensive for GPUs.

6.4 Ray-Tracing Results

The multiview rendering of our ray-tracing algorithm did not result in real-time performance. Tracing the depth map turned out to be memory intensive. This becomes a problem when scaling to high definition output resolution. Fetching over 16 pixels brings down the performance to only two frames per second. These texture fetches are required for supporting up to 16 pixels of horizontal disparity, which is not that big for a 3D impression. The results are shown in figure 55.

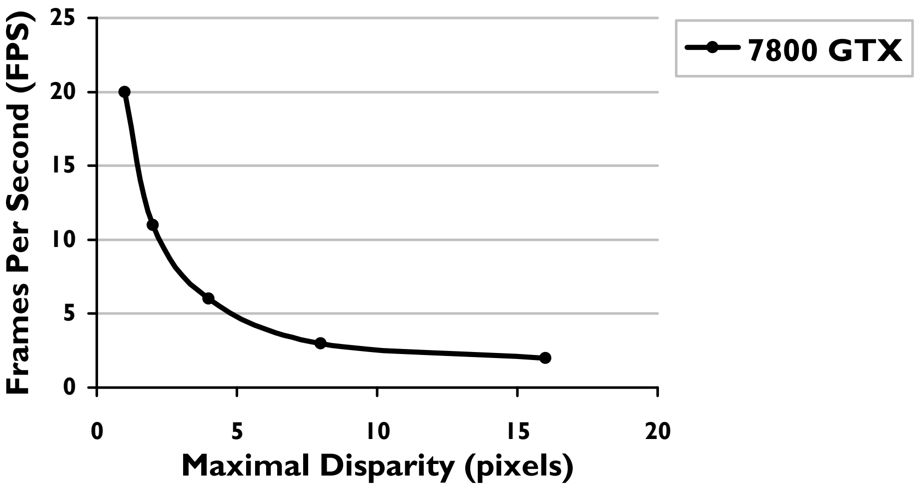

Figure 55. Ray-tracing on the pixel shader is memory bandwidth limited. Doubling the one-dimensional search range about halves the frame rate (FPS), because twice as many texture samples are needed.

7 Conclusion

We have presented a method for multiview rendering on a GPU. Our application proves that modern personal computers equipped with NVIDIA's 7800 GTX or comparable video card are powerful enough for nine-view rendering. Comparable, in this case, means the same number of vertex and pixel shaders. The computational power per shader should also be identical. This property is reflected by the support shader model version. The clock frequency of the GPU and its memory also influences performance.

To be more specific, nine-view rendering requires at least 12 fragment processors to merge these views to the lenticular format at 1920 × 1080 output pixels. The 7800 GTX has 24 fragment processors, but these are loaded about half. This is deduced from the fact that we were able to sustain a little over 50 frames per second throughput, when minimizing the workload on the vertex shaders, as shown in section 6.2. This frame rate is twice the target of 25 frames per second, the latter which we consider as real-time performance. So, effectively about half the number of fragment processors should suffice.

As mentioned in section 2.3, the fragment pipes also consist of raster operators, not only fragment processors. Some GPUs have less raster operators than they have fragment processors. This is less of a concern to us, because the render engine does not rely on the special features of these raster operators.

The micropolygon displacement mapping algorithm typically involves a lot of triangle processing. A grid of 720 × 576 cells holds about 0.8 million triangles and requires eight vertex processors to do its math. So, current vertex units are good for processing about 25 × 0.1 million triangles each second. In section 6.1 we showed that – of the three video cards we tested – only the 7800 GTX is powerful enough to handle these computations in real-time. We could reduce the number of cells to sustain real-time speeds on lesser specified GPUs, but this sacrifices some of the quality of the 3D impression.

Real-time multiview rendering both requires powerful pixel processing and vertex processing. Interleaving a set of views conveniently maps to the pixel shader hardware. Many image processing algorithms do. However, the vertex shader hardware can also be put to good use, even for image manipulation. For example, our algorithm generates disparity by deforming textured geometry on de vertex shader hardware. In fact, the vertex shader hardware is a limiting factor when rendering at full resolution. The performance for micropolygon displacement mapping already starts to decline at 90 × 72 cells, as the graph in section 6.1 shows. A solution is to effectively trade vertex processing for fragment processing by undersampling the depth map on the vertex shader and texture correct the resulting artifacts on the pixel shader. The framework for this approach has already been laid out in section 4.5.

We are able to render RGBD sources to a lenticular display at a 25 frames per second throughput. Programmable GPUs are not limited to lenticular display rendering, but are also suitable for driving other types of multiview devices. These programmable video cards extend the PC to a valuable development platform for video processing. However, not all algorithms lend itself for a efficient GPU implementation, as we experienced with our ray-tracing attempt. Strictly speaking, expensive ray-tracing is only needed for occlusion and de-occlusion support. Course horizontal shifts to generate disparity are possible without sampling each pixel by exploiting the vertex shader hardware. We proved this by implementing the micropolygon shader. To make depth map tracing into a real-time solution for multiview rendering, we could restrict the tracing process to areas near object borders. However, object (border) recognition in image sequences is yet another difficult problem, but we did already succeeded in detecting and visualizing problem areas. So, programmable GPUs also help the developers to debug their computer graphic solutions.

We conclude that at least a 3.0 GHz personal computer and video card with 8 programmable vertex pipes and 24 programmable fragment pipes are a suitable starting point for real-time nine-view rendering at high definition resolution. The load on the vertex shaders can be maximized by tuning the triangle grid. Interleaving subpixels typically puts a load on the pixel shader hardware. So, the 24 pixel pipes are mostly determine the feasible output resolutions.

Increasing the number of views beyond nine requires even more processing power. To keep the same quality the number of grid triangles has to be increased proportionally. So, more vertex processing needs to be done. The current GPUs on the consumer market have eight vertex processors at most, which poses a problem. One solution is to use multiple GPUs. For example, we expect a dual 7800 or 7800 in SLI mode to be sufficient to drive a 15-view display device. The total number of fragment processor in such a set-up is also sufficient to multiplex the views to the 1920 × 1080 output resolution. Scaling beyond four GPUs is yet impossible, given today's state of technology. Current motherboards support two single or two dual GPUs (called quad SLI) at most. The latter configuration could also enable output to quad HD screens.

8 Recommendations

DirectX is a suitable API for exploiting the latest technologies on computer graphics. However we did encounter some stability issues during development, including hardware resets. This could be due to driver problems and caused by misuse of certain DirectX calls. The graphics runtime seems to be sensitive to invalid resource creation in video memory and certain malformed shader programs.

In the near future we will see unified shader hardware. This relaxation will be supported starting from DirectX version 10, which will be incorporated in Windows Vista. It will allow designers to trade vertex processing for pixel processing and vice versa. This could help us in improving our micropolygon displacement shader. Other massively parallel computations also benefits from the ongoing development of shader technology. We already see dual-core CPU and dual GPU systems and this trend of more parallelism will undoubtedly continue. However, it will take extra programming effort to exploit these increasingly parallel systems.

The current implementation of micropolygon displacement shader has a few minor deficiencies. The most apparent deficiency occurs when the disparity effect is set to higher values. In this case artifacts arise at the left and right side of the output image. These artifacts are caused by the 2×2 layout of the tiles in the texture atlases. The texture mapped polygons of two horizontally adjacent views invade the orthogonal projected space of one another. We can resolve this by choosing a 4×1 atlas layout or by inserting space between the tiles. We already experimented with the latter solution, but it has a small impact on performance, due to additional shader code for texture addressing. The former solution does not suffer from this disadvantage.

Another deficiency is caused by the triangular structure of the texture mapped grid. These artifacts become visible when using a course grain grid. Ideally, triangles are as small as the texels. The performance of current graphics hardware is close to having true micropolygons, but in the mean time we hope to work out a texture correction scheme to overcome this problem.

Yet another improvement would be extending the DirectX framework to support pluggable shaders similar to NVIDIA's FX Composer. We do not intend to create a full integrated development environment, but a specialized tool could ease the development of new shaders for video processing. The focus of FX Composer is on supporting shaders for 3D game graphics, which excludes video sources.

We can also add support for different kinds of multiview display devices. The view creation and its interleaving are deliberately separated over distinct shaders programs. This means we can write new interleaving code or expand the number of views independently. However additional views will require more processing power. For example, a 15-view display will not only need more video memory, but also more powerful vertex and fragment processing. The size of the triangle grid is related to the input resolution. Larger output sizes require more fragment processing and video memory bandwidth. The shader hardware requirements scale linearly with the number of views. Because of the ongoing developments programmable GPUs will eventually support additional views.

Finally, a small adjustment is to improve file handling. The currently used library (CPFSPD[*]) is not particularly suitable for streaming RGBD, due to heavy disk seeks. We could implement a system memory buffering algorithm to save on expensive disk arrays (RAID). This will also improve the frame stepping responsiveness of our application.

[*] CPFSPD is a cross-platform C-library implementing the Philips File Standard for Pictorial Data.

A ATI Specifications

Radeon Model	9200	9600	9800	X550	X600	X700	X800	X850
Bus Speed AGP/PCIe	1×/-	8×/-	8×/-	-/×16	8×/×16	8×/×16	8×/×16	8×/×16
Max. Memory	128MB	128MB	128MB	128MB	256MB	256MB	256MB	256MB
Memory Interface	128-bit	128-bit	128-bit	128-bit	128-bit	128-bit	256-bit	256-bit
Vertex Pipes	2	2	4	2	2	6	6	6
Vertices/sec	62.5M	1.3M	325M	?	?	637M*	780M*	810M*
Pixel Pipes	4	4	8	4	4	8	12-16	12-16
Pixels/sec	1.0G	1.3G	2.6G	1.6G	2.0G*	3.4G*	8.3G*	8.6G*
Texture units	4	4	8	4	4	8	8-16	16
Textures/pass	6	16	16	16	16	16	16	16
DirectX	8.1	9.0	9.0	9.0	9.0	9.0	9.0b	9.0b
Shader Model	1.4	2.0	2.0	2.0	2.0	2.0b	2.0b	2.0b
OpenGL	1.3+	1.5+	2.0+	1.5+	2.0+	2.0+	2.0	2.0

*When using the fastest bus type (PCI Express).
+Functionality exposed via OpenGL Extensions.

FireGL Model	T2-128	Z1-128	X2-256t	X3-256	V3100	V3200	V5000	V5100	V7100
Bus Speed AGP/PCIe	8×/-	8×/-	8×/-	8×/-	-/×16	-/×16	-/×16	-/×16	-/×16
Core	9600	9500	9800XT	X800Pro	X300	X600	X700	X800Pro	X800XT
Max. Memory	128MB	128MB	256MB	256MB	128MB	128MB	128MB	128MB	256MB
Memory Interface	128-bit	256-bit	256-bit	256-bit	128-bit	128-bit	128-bit	256-bit	256-bit
Vertex Pipes	2	4	4	6	2	2	6	6	6
Vertices/sec	200M	300M	412M	750M	200M	250M	637M	675M	750M
Pixel Pipes	4	4	8	12	4	4	8	12	16
Pixels/sec	1.6G	1.3G	3.3G	5.4G	1.6G	2.0G	3.4G	5.4G	8.0G
Texture Units	4	4	8	12	4	4	8	12	16
Textures/pass	16	16	16	16	16	16	16	16	16
DirectX	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0
Shader Model	2.0	2.0	2.0	2.0b	2.0	2.0	2.0b	2.0b	2.0b
OpenGL	1.5+	1.5+	1.5+	1.5+	1.5+	1.5+	1.5+	1.5+	1.5+

+Functionality exposed via OpenGL Extensions.

(These tables were compiled from http://www.ati.com/products/workstation/fireglmatrix.html and ATI's product specification site as of July 18th 2005.)

B Matrox Specifications

Matrox Model	**G450**	**G550**	**P650**	**P750**	**Parhelia**
Bus Speed AGP/PCIe	4×/-	4×/-	8×/-	8×/-	8×/-
Max. Memory	32 MB	32 MB	64 MB	64 MB	256 MB
Memory Interface	64-bit	64-bit	128-bit	128-bit	256-bit
Vertex Pipes	0	0	2	2	4
Vertices/sec					
Pixel Pipes	2	2	2	2	4
Pixels/sec					
Texture Units	1+1	2+2	4	4	16
Textures/pass					4
DirectX	6	6	8.1	8.1	8.1
OpenGL	1.1	1.1	1.3	1.3	1.3

(This table was constructed from the Matrox product site as of July 18th 2005.)

C NVIDIA Specifications

NVidia Model	5200	5600	5800	5900	6200	6600	6800	7800
GPU Code	NV34	NV31	NV30	NV35	NV44	NV43	NV40	G70
Bus Speed AGP/PCIe	8×/-	8×/-	8×/-	8×/-	8×/×16	8×/×16	8×/×16	-/×16
Max. Memory	256 MB	256 MB	256 MB	256 MB	256 MB	256 MB	512 MB	≤256MB
Memory Interface	64/128-bit	64/128-bit	128-bit	256-bit	128-bit	128-bit	256-bit	256-bit
Vertex Pipes	2	2	3	3	2	3	6	8
Vertices/sec	81M	88M	160M	338M	263M*	375M*	600M*	860M
Pixel Pipes	4	4	4	4	4	8	12-16	24
Pixels/sec	1.3G	1.6G	1.6G	1.6G	1.4G*	4.0G*	6.4G*	10.3G
ROPs	4	4	4	4	2-4	4	12-16	16
Texture Units	4	4	8	8	4	8	12-16	32
Textures/pass	4×2	2×2 or 4×1	8×1	4×2	16	16	16	16
DirectX	9.0	9.0	9.0	9.0	9.0c	9.0c	9.0c	9.0c
Shader Model	2.0	2.0	2.0	2.0	3.0	3.0	3.0	3.0
OpenGL		1.4	1.5	1.4	1.5	1.5	1.5	2.0

*When using the fastest bus type (PCI Express).

(This table was constructed from NVIDIA's product site and various specification PDFs as of July 28th 2005.)

D Shader Models

Vertex shader version	**vs_1_1**	**vs_2_0**	**vs_2_x**	**vs_3_0**
Maximum instructions	128	256	256	>512
Address registers (a#)	1	1	1	1
Input registers (v#)	16	16	16	16
Temporary registers (r#)	12	12	2 (12)	32
Constant boolean registers	0	16	16	16
Constant float registers	2 (96)	2 (256)	3 (256)	2 (256)
Constant integer registers	0	16	16	16
Loop counter registers (aL)	0	1	1	1
Predicate registers (p)	0	0	0	1
Texture samplers (s#)	0	0	0	4
Texture coordinates (oT#)	8	8	8	
Output (oPos+oFog+oPts)	1+1+1	1+1+1	1+1+1	12
Color registers (oD1, oD2)	2	2	2	

(Source: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/assemblylanguageshaders/vertexshaders/vertexshaders.asp)

Pixel shader version	**ps_1_1**	**ps_1_2**	**ps_1_3**	**ps_1_4**	**ps_2_0**	**ps_2_x**	**ps_3_0**
Maximum instructions (texture+arithmetic)	4+8	4+8	4+8	6+8	32+64	>96	>512
Input color registers (v#)	2	2	2	2	2	2	10
Temporary registers (r#)	0	0	0	0	12-32	12-32	32
Constant boolean registers	0	0	0	0	0	16	16
Constant float registers	0	0	0	0	32	32	224
Constant integer registers	0	0	0	0	0	16	16
Loop counter register (aL)	0	0	0	0	0	1	1
Predicate register (p)	0	0	0	0	0	1	1
Texture samplers (s#)	4	4	4	8	16	16	16
Texture coordinates (t#)	0	0	0	0	8	8	8
Dynamic flow control IF x>5	No	No	No	No	No	Yes	Yes
Static flow control IF x=5	No	No	No	No	No	Yes	Yes
Arbitrary source swizzle	No	No	No	No	No	Yes	Yes
Gradient instructions	No	No	No	No	No	Yes	Yes
Prediction	No	No	No	No	No	Yes	Yes

(Source: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/assemblylanguageshaders/pixelshaders/pixelshaders.asp)

E Tabularized Test Results

The following tables provide our test measurements in numbers, which were used for the graphs found in chapter 6.

These are the frame rates when varying the number of cells of the texture mapped geometry and fixing the output resolution at 1920 × 1080 pixels:

Grid Cell Count	**6200**	**6600 GT**	**7800 GTX**
90 × 72	5	18	58
180 × 144	5	17	56
240 × 192	5	16	53
360 × 288	5	14	42
540 × 432	3	10	31
720 × 576	3	8	23

The next table lists the frame rates when varying the output resolution and using a grid of 3 × 2 cells:

Output Resolution	**6200**	**6600 GT**	**7800 GTX**
512 × 384	16	93	240
720 × 576	12	65	180
1024 × 768	9	41	120
1280 × 1024	7	28	80
1600 × 1200	5	20	58
1920 × 1080	5	18	55

F YC_bC_r to RGB Conversion

The following formulas define the implemented YC_bC_r (conform signal encoding standard CCIR 601) to RGB conversion as proposed by Microsoft [75]:

$$C = Y - 16$$
$$D = C_b - 128$$
$$E = C_r - 128$$

This defines three integer coefficients that are used to compute the RGB values:

$$R = ((298 \times C) + (409 \times E) + 128) >> 8$$
$$G = ((298 \times C) - (100 \times D) - (208 \times E) + 128 >> 8$$
$$B = ((298 \times C) + (516 \times D) + 128) >> 8$$

The operator $>>$ is the bit-shift-right operator. Only the eight most significant bits are used of each component. Some precision is lost using integer math.

Each component R, G and B needs to be clamped to the range $[0,255]$.

More information can be found at http://www.fourcc.org/fccyvrgb.php

G References

1 Wikipedia community. 2006. Broadcast Television Systems. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Broadcast_television_system

2 C. Poynton. 2003. Introduction. In *Digital Video and HDTV Algorithms and Interfaces*, pp. 3—127.

3 Wikipedia community. 2006. Advanced Technology Attachment. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Advanced_Technology_Attachment

4 Silicon Integrated Systems Corp. 2005. The Evolution of Chipset I/O Interface and PCI Express. In *The PCI Express Technology*, http://www.sis.com/elibrary/elibrary_index00_000011.htm

5 Wikipedia community. 2005. PCI Express. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/PCI_Express

6 M. Chambers. 1999. GPU Overview. In *NDIVIA GeForce 256*, http://www.nvnews.net/reviews/geforce_256/preface.shtml

7 NVIDIA Press Release. 2001. In *NVIDIA Introduces GeForce3 for the PC*, http://www.nvidia.com/object/IO_20010530_5676.html

8 W. Harris. 2005. Some History. In *A bluffer's guide to Shader Models*, http://www.bit-tech.net/hardware/2005/07/25/guide_to_shaders/1.html

9 NVIDIA Press Release. 2002. In *NVIDIA GeForce FX GPU Ushers in a New Era of Cinematic Computing*, http://www.nvidia.com/object/IO_20021117_7139.html

10 D. Luebke. 2003. Classic Rendering Pipeline. In *The GPU Revolution: Programmable Graphics Hardware*, p. 1—2, http://www.cs.virginia.edu/~gfx/Courses/2003/Intro.fall.03/slides/gpu_web/gpu.pdf

11 I. E. Sutherland. 1974. Reentrant polygon clipping. In *Communications of the ACM Vol. 7 Num. 1*, p. 32.

12 R. F. Puk. 1977. General clipping on an oblique viewing frustum. In *ACM SIGGRAPH Computer Graphics, Vol. 11 Issue 2*.

13 Microsoft Corp. 2005. Rendering from Vertex and Index Buffers. In *Rendering Primitives*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/GettingStarted/Rendering/renderingprimitives/renderingfromvertexindexbuffers.asp

14 ATI Corp. 2005. Vertex Processing Engine. In *ATI Demos*, http://mirror.ati.com/vortal/r350/flash/9800educational/index.html

15 P. Gerasimov, R. Fernando, S. Green. 2004. Using Vertex Fetches. In *Shader Model 3 Whitepaper*, http://developer.nvidia.com/object/using_vertex_textures.html

16 Wikipedia community. 2005. Primitive-by-primitive. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Rendering

17 ExtremeTech. 2004. Making the Trains Run On Time. In *Preview: NVIDIA's GeForce 6800 Ultra*, http://www.extremetech.com/article2/0,1558,1567090,00.asp

18 Wikipedia community. 2005. Z-buffering. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Z-buffer

19 NVIDIA Corp. 2005. Introduction. In *Improving Shadows and Reflections via the Stencil Buffer*, p. 2, http://developer.nvidia.com/object/Stencil_Buffer_Tutorial.html

20 S. Dietrich. 1999. Game Developers Conference presentation. On *Using the Stencil Buffer*, p. 3, http://developer.nvidia.com/attach/7294

21 W. R. Mark, K. Proudfoot. 2001. Compiling to a VLIW Fragment Pipeline. In *Proceedings of 2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, p. 2.

22 The Tech Report. 2004. The GeForce 6200. In *NVIDIA's GeForce 6200 graphics processor*, p. 1, http://techreport.com/reviews/2004q4/geforce-6200/

23 The Tech Report. 2004. The NV44 GPU. In *NVIDIA's GeForce 6200 with TurboCache*, p. 1, http://techreport.com/reviews/2004q4/geforce-6200-turbocache/

24 Wikipedia community. 2006. Common Display Resolutions. In *Display resolutions*, http://en.wikipedia.org/wiki/Display_resolution

25 NVIDIA Corp. 2005. Chapter 4.8. In *GPU Programming Guide 2.4.0*, p. 40—41, http://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf

26 NVIDIA Corp. 2005. Introduction. In *TurboCache Technical Brief*, http://www.nvidia.com/object/IO_17361.html

27 ATI Technologies Inc. 2005. Multiply & Conquer. In *CrossFire Brochure*, http://www.ati.com/technology/crossfire/CrossFireBrochure.pdf

28 ATI Technologies Inc. 2005. What is H.264? In *H.264 Whitepaper*, http://www.ati.com/products/pdf/H264_Whitepaper.pdf

29 E. A. Catmull. 1974. Computer Display of Curved Surfaces. In *Proceedings of Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pp. 11—17.

30 P. Heckbert. 1986. Uses for Texture Mapping. In *Survey of Texture Mapping*, p. 2.

31 J. J. Koendrink, A. J. V. Doorn. 1996. Illuminance Texture Due to Surface Mesostructure. *Journal of the Optical Society of America 13*, 3, pp. 452—463.

32 J. Hastings-Trew. 2005. Creating Normal Maps with Cinema 4-D (part 1), http://members.shaw.ca/jimht03/normal.html

33 J. Blinn. 1978. Simulation of Wrinkled Surfaces. In *Proceedings of the 5th Conference on Computer Graphics*.

34 M. Oliveira, G. Bishop, D. McAllister. 2000. Relief Texture Mapping. In *Proceedings of SIGGRAPH*, pp. 359—368.

35 N. Max. 1988. Horizon Mapping: shadows for Bump-Mapped Surfaces. In *The Visual Computer 4*, 2, pp. 109—117.

36 P.-P. Sloan, M. F. Cohen. 2000. Interactive Horizon Mapping. In *Eurographics Workshop on Rendering*, pp. 175—186.

37 W. Heidrich, K. Daubert, J. Kautz, H.-P. Seidel. 2000. Illuminating Micro Geometry Based on Precomputed Visibility. In *SIGGRAPH '00 Proceedings of Computer Graphics*, pp. 455—464.

38 R. Cook. 1984. Shade Trees. In *Proceedings of SIGGRAPH*, pp. 223—231.

39 R. Cook, L. Carpenter, E. Catmull. 1987. The Reyes Image Rendering Architecture. In *Proceedings of SIGGRAPH*, pp. 95—102.

40 S. Dietrich. 2000. Elevation Maps. In *Technical Report from NVIDIA Corp.*, pp. 1—12.

41 J. Kautz, H.-P. Seidel. 2001. Hardware Accelerated Displacement Mapping for Image Based Rendering. In *Graphics Interface*, pp. 61—70.

42 J. W. Patterson, S. G. Hoggar, J. R. Logie. 1991. Inverse Displacement Mapping. In *Computer Graphics Forum Volume 10 (2)*, p. 129—139.

43 J. R. Logie, J. W. Patterson. 1995. Inverse Displacement Mapping in the General Case. In *Computer Graphics Forum Volume 14 (5)*, pp. 261—273.

44 B. Smits, P. Shirley, M. M. Stark. 2000. Direct Ray Tracing of Displacement Mapped Triangles. In *Eurographics Workshop on Rendering*, pp. 307—318.

45 E. W. Weisstein. 2006. Barycentric Coordinates. In *MathWorld–A Wolfram Web Resource*, http://mathworld.wolfram.com/BarycentricCoordinates.html

46 F. K. Musgrave, C. E. Kolb, R. S. Mace. 1989. The Synthesis and Rendering of Eroded Fractal Terrains. In *16th Proceedings on Computer Graphics and Interactive Techniques Volume 23 (3)*, pp. 41—50.

47 L. McMillan. 1997. An Image-Based Approach to Three-Dimensional Computer Graphics. In *PhD thesis, University of Noth Carolina at Chapel Hill.*

48 G. Schaufler, M. Priglinger. 1999. Efficient Displacement Mapping by Image Warping. In *10th Eurographics Rendering Workshop*, pp. 183—194.

49 L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B, Guo, H.-Y. Shum, 2003, View-Dependent Displacement Mapping. In *ACM Transactions on Graphics Volume 22 (3)*, pp. 334—339.

50 A. Rosenfeld, J. L. Pfalz. Distance Function on Digital Pictures. In *Pattern Recognition Vol. 1*, pp. 33–61.

51 Y.-H. Lee, S.-J. Horng, J. Seitzer. 2003. Parallel Computation of the Euclidean Distance Transform on a Three-Dimensional Image Array. In *IEEE Transactions on Parallel and Distributed Systems, Vol 14 (3)*, pp. 203–213.

52 E.W. Weisstein. 2006. Affine Space. In *MathWorld--A Wolfram Web Resource.* http://mathworld.wolfram.com/AffineSpace.html

53 A. Finkelstein. 2001. Parallel Projection. In *3D Polygon Rendering Pipeline*, pp. 26.

54 O. Tolba, J. Dorsey, L. McMillan. 2001. A Projective Drawing System. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 27.

55 C. van Berkel, J.A. Clarke. 1997. Characterisation and Optimisation of 3D-LCD Module Design. In *Proceedings of SPIE Vol. 3012*, pp. 179–187.

56 C. van Berkel. 1999. Image Preparation on 3D-LCD. In *Proceedings of SPIE Volume 3639 Stereoscopic Displays and Vitual Reality Systems VI.*

57 F. M. Cardocia, J. C. Principe. Introduction. In *Superresolution of Images with Learned Multiple Reconstruction Kernels*, pp. 2.

58 Wikipedia community. 2006. Convolution theorem. In *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Convolution_theorem

59 ExtremeTech. 2002. ATI vs NVIDIA: Adaptive Filtering. In *The Naked Truth About Anisotropic Filtering*, p. 3, http://www.extremetech.com/article2/0,3973,548248,00.asp

60 BeHardware. 2004. Anisotropic Filtering. In *ATI Radeon X800 XT and X800 Pro*, p. 6, http://www.behardware.com/articles/494-6/ati-radeon-x800-xt-and-x800-pro.html

61 H. S. Hou, H. C. Andrews. 1978. Cubic splines for image interpolation and digital filtering. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 508—517.

62 J. A. Parker, R.V. Kenyon, D. E. Troxel. 1983. Resampling. In *Comparison of Interpolating Methods for Image Resampling*, p. 36.

63 W. Donnelly. 2005. Per-Pixel Displacement Mapping with Distance Function. In *GPU Gems 2*, http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch08.pdf

64 M. Christen. 2005. Ray Tracing on GPU. In *Thesis from University of Applied Sciences Basel.*

65 T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan. 2002. Ray Tracing on Programmable Graphics Hardware. In *Proceedings of the 29th annual conference on computer graphics and interactive techniques SIGGRAPH 2002.*

66 W. Jones. 2002. The What, Why, and How of DirectX. In *Beginning DirectX*, p. 6.

67 I. Buck. 2004. Abstract. In *Brook for GPUs: Stream Computing on Graphics Hardware*, http://graphics.stanford.edu/papers/brookgpu/

68 NVIDIA Corp. 2006. Introduction to the Cg Language. In *User's Manual*, pp. 1—32, http://download.nvidia.com/developer/cg/Cg_1.4/1.4.1/Cg-1.4.1_UsersManual.pdf

69 J. Kessenich, D. Baldwin, R. Rost. 2004. Basic Types. In *The OpenGL Shading Language*, pp. 16, http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf

70 A. Lovesey. 2005. A Comparison of Real Time Graphical Shader Languages. In *CS4983 Senior Technical Report*, pp. 1—42, http://www.cs.unb.ca/undergrad/html/documents/Lovesey_Senior_TechReport.pdf

71 C. Simonyi. 1999. Hungarian notation reprint. In *MSDN Library*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp

72 J. Spolsky. 2005. Making Wrong Code Look Wrong. In *Joel on Software*, http://www.joelonsoftware.com/articles/Wrong.html

73 P. Gerasimov, R. Fernando, S. Green. 2004. Using Vertex Fetches. In *Shader Model 3 Whitepaper*, http://developer.nvidia.com/object/using_vertex_textures.html

74 M. Pharr. 2005. Fast Third-Order Texture Filtering. In *GPU Gems 2*, pp. 313—328.

75 G. Sullivan, S. Estrop. 2002. Video Rendering with 8-bit YUV Formats. In *Microsoft's MSDN Library*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwmt/html/yuvformats.asp